UNIVERSIDAD POLITÉCNICA DE MADRID

Escuela Universitaria de Ingeniera Técnica de Telecomunicación

INTEGRACIÓN MPLAYER – OPENSVC EN EL PROCESADOR MULTINÚCLEO OMAP3530

TRABAJO FIN DE MÁSTER

Autor: Óscar Herranz Alonso

Ingeniero Técnico de Telecomunicación

Tutor:

Fernando Pescador del Oso

Doctor Ingeniero de Telecomunicación

Julio 2012

Máster en Ingeniería de Sistemas y Servicios para la Sociedad de la Información

Trabajo Fin de Máster		
Título	Integración Mplayer - OpenSVC en el procesador multinúcleo OMAP3530	
Autor	D. Óscar Herranz Alonso	Vº Bº
Tutor	D. Fernando Pescador del Oso	
Ponente		
Tribunal		
Presidente	D. Alfonso Martín Marcos	
Secretario	D. Matías Garrido González	
Vocal	D. Angel Groba González	
Fecha de lectura		
Calificación		

El Secretario:

AGRADECIMIENTOS

Un año y pocos meses después de la defensa de mi Proyecto Fin de Carrera me vuelvo a encontrar en la misma situación: escribiendo estás líneas de mi Trabajo Fin de Máster para agradecer a aquellas personas que me han apoyado y ayudado de alguna forma durante esta etapa de mi vida. Después de un año en el que he hecho demasiadas cosas importantes en mi vida (y sorprendentemente todas bien), ha llegado la hora de dar por finalizado el Máster, el último paso antes de cerrar mi carrera de estudiante.

Por ello, me gustaría agradecer en primer lugar al Grupo de Investigación GDEM por darme la oportunidad de formar parte de su equipo y en especial a Fernando, persona ocupada donde las haya, pero que una vez más me ha aconsejado en numerosas ocasiones el camino a seguir para solventar los problemas. Gracias Fernando por tu tiempo y dedicación.

Agradecer a mis padres, Fidel y Victoria, y a mi hermano, Víctor, el apoyo y las fuerzas recibidas en todo momento. Sé que no todos podréis estar presentes en mi defensa de este Trabajo Fin de Máster pero da igual. Ya me habéis demostrado con creces lo maravillosos que sois. Gracias por vuestro apoyo incondicional y por recibirme siempre con una sonrisa dibujada en vuestro rostro.

Por supuesto, no me puedo olvidar de Ruth, artísticamente conocida como RuthGe ;), la cual me ha vuelto a demostrar su encanto a lo largo de esta etapa en la que no he podido dedicarla todo el tiempo que hubiera querido. Gracias por estar siempre ahí y apoyarme, por los viajes hechos y por hacer, por tantas cosas que me hacen sentir especial cuando estoy a tu lado. Te prometo que nos espera un verano por delante espectacular.

También quiero mencionar, como no, a Miguel, pese a que más de medio Máster haya estado a kilómetros de distancia, no ha sido ni mucho menos suficiente para seguir manteniendo nuestros "cotis" y risas varias. Y los que nos quedan… Merci de m'accueillir dans votre maison à Rennes.

A mi amigo Felipe, que por fin se ha hecho un hombre ingeniero, terminando su carrera como los mejores. Aunque no nos hemos visto todo lo que hubiéramos querido durante el invierno, sabemos que nuestra etapa fuerte es el veranito y las charlas de piscina (y este año pádel). Ya tengo ganas de una de esas.

Tampoco quiero olvidarme de mis amigos "calladitos" pero grandes personas, como Quique y Rubén. Este año ha sido el primero que nuestro caminos se han separado y

la verdad que echo de menos los viajes ida y vuelta a la universidad hablando de fútbol y de "otras cosas" que no puedo mencionar aquí. Pero lo importante es que seguimos manteniendo la amistad y el buen rollo entre nosotros.

Y por último, pero no por ello menos importante, quiero agradecer a mis amigos "no ingenieros" Navares y Tomás, los grandes momentos vividos durante este año sobre todo de noche. No sabéis lo feliz que me hace seguir manteniendo una relación tan buena con vosotros, con "mis amigos de toda la vida". Gracias por tantas noches de fiesta, risas y cachondeo general que me han servido para desconectar lo suficiente y no terminar paranoico con temas ingenieriles.

Gracias a todos de corazón.

ÍNDICE GENERAL

RELACIÓN DE FIGURAS

RELACIÓN DE TABLAS

RESUMEN

La constante evolución de dispositivos portátiles multimedia que se ha producido en la última década ha provocado que hoy en día se disponga de una amplia variedad de dispositivos con capacidad para reproducir contenidos multimedia. En consecuencia, la reproducción de esos contenidos en dichos terminales lleva asociada disponer de procesadores que soporten una alta carga computacional, ya que las tareas de descodificación y presentación de video así lo requieren.

Sin embargo, un procesador potente trabajando a elevadas frecuencias provoca un elevado consumo de la batería, y dado que se pretende trabajar con dispositivos portátiles, la vida útil de la batería se convierte en un asunto de especial importancia.

La problemática que se plantea se ha convertido en una de las principales líneas de investigación del Grupo de Investigación GDEM (Grupo de Diseño Electrónico y Microelectrónico). En esta línea de trabajo, se persigue cómo optimizar el consumo de energía en terminales portables desde el punto de vista de la reducción de la calidad de experiencia del usuario a cambio de una mayor autonomía del terminal.

Por tanto, para lograr esa reducción de la calidad de experiencia mencionada, se requiere un estándar de codificación de vídeo que así lo permita. El Grupo de Investigación GDEM cuenta con experiencia en el estándar de vídeo escalable H.264/SVC, el cual permite degradar la calidad de experiencia en función de las necesidades/características del dispositivo. Más concretamente, un video escalable contiene embebidas distintas versiones del video original que pueden ser descodificadas en diferentes resoluciones, tasas de cuadro y calidades (escalabilidades espacial, temporal y de calidad respectivamente), permitiendo una adaptación rápida y muy flexible.

Seleccionado el estándar H.264/SVC para las tareas de vídeo, se propone trabajar con Mplayer, un reproductor de vídeos de código abierto (*open source*), al cual se le ha integrado un descodificador para vídeo escalable denominado OpenSVC.

Por último, como dispositivo portable se trabajará con la plataforma de desarrollo BeagleBoard, un sistema embebido basado en el procesador OMAP3530 que permite modificar la frecuencia de reloj y la tensión de alimentación dinámicamente reduciendo de este modo el consumo del terminal. Este procesador a su vez contiene integrados un procesador de propósito general (ARM Cortex-A8) y un procesador digital de señal (DSP TMS320C64+TM).

Debido a la alta carga computacional de la descodificación de vídeos escalables y la escasa optimización del ARM para procesamiento de datos, se propone llevar a cabo la ejecución de Mplayer en el ARM y encargar la tarea de descodificación al DSP, con la finalidad de reducir el consumo y por tanto aumentar la vida útil del sistema embebido sobre el cual se ejecutará la aplicación desarrollada.

Una vez realizada esa integración, se llevará a cabo una caracterización del descodificador alojado en el DSP a través de una serie de medidas de rendimiento y se compararán los resultados con los obtenidos en el proceso de descodificación realizado únicamente en el ARM.

ABSTRACT

During the last years, the multimedia portable terminals have gradually evolved causing that nowadays a several range of devices with the ability of playing multimedia contents are easily available for everyone. Consequently, those multimedia terminals must have high-performance processors to play those contents because the coding and decoding tasks demand high computational load. However, a powerful processor performing to high frequencies implies higher battery consumption, and this issue has become one of the most important problems in the development cycle of a portable terminal.

The power/energy consumption optimization on multimedia terminals has become in one the most significant work lines in the Electronic and Microelectronic Research Group of the Universidad Politécnica de Madrid. In particular, the group is researching how to reduce the user's Quality of Experience (QoE) quality in exchange for increased battery life.

In order to reduce the Quality of Experience (QoE), a standard video coding that allows this operation is required. The H.264/SVC allows reducing the QoE according to the needs/characteristics of the terminal. Specifically, a scalable video contains different versions of original video embedded in an only one video stream, and each one of them can be decoded in different resolutions, frame rates and qualities (spatial, temporal and quality scalabilities respectively).

Once the standard video coding is selected, a multimedia player with support for scalable video is needed. Mplayer has been proposed as a multimedia player, whose characteristics (open-source, enormous flexibility and scalable video decoder called OpenSVC) are the most suitable for the aims of this Master Thesis.

Lastly, the embedded system BeagleBoard, based on the multi-core processor OMAP3530, will be the development platform used in this project. The multimedia terminal architecture is based on a commercial chip having a General Purpose Processor (GPP – ARM Cortex A8) and a Digital Signal Processor (DSP, TMS320C64+™). Moreover, the processor OMAP3530 has the ability to modify the operating frequency and the supply voltage in a dynamic way in order to reduce the power consumption of the embedded system.

So, the main goal of this Master Thesis is the integration of the multimedia player, MPlayer, executed at the GPP, and scalable video decoder, OpenSVC, executed at the DSP in order to distribute the computational load associated with the scalable video decoding task and to reduce the power consumption of the terminal.

Once the integration is accomplished, the performance of the OpenSVC decoder executed at the DSP will be measured using different combinations of scalability values. The obtained results will be compared with the scalable video decoding performed at the GPP in order to show the low optimization of this kind of architecture for decoding tasks in contrast to DSP architecture.

1 INTRODUCCIÓN Y OBJETIVOS

En este capítulo se realiza una introducción a este Trabajo Fin de Máster, describiendo brevemente el contexto en el que se encuadra el trabajo desarrollado, así como los dos elementos principales empleados para su materialización: el sistema embebido BeagleBoard basado en el procesador OMAP3530 y el estándar de codificación de vídeo escalable SVC.

A continuación, se define el objetivo principal del trabajo realizado, desglosando éste en una serie de hitos de menor magnitud y ordenados temporalmente.

Por último, se realiza una descripción de la organización de la presente memoria, explicando de forma sintetizada los diferentes temas abordados en cada capítulo, así como los anexos, referencias bibliográficas.

1.1 Introducción

En los últimos años se ha producido un notable incremento de dispositivos portables (teléfonos móviles, *smartphones*, *tablets PC,* etc.) con capacidad de visualizar contenidos multimedia de diferentes calidades, siendo la mayoría de ellos asequibles para cualquier Todos estos dispositivos son lo que se conoce como sistema embebido, esto es, un sistema de computación con un alto nivel de integración *hardware* dedicado a una aplicación específica. Generalmente, está formado por una unidad que aporta capacidad de cómputo al sistema (un procesador con una arquitectura genérica), y un conjunto de componentes dependientes del propio sistema embebido: memorias, interfaces de comunicación, aceleradores *hardware*, sensores, etc. Hoy en día, el uso de los sistemas embebidos está creciendo exponencialmente dadas sus características y su campo de aplicación es amplísimo: consumo, comunicaciones, automoción o robótica son sólo algunas de las áreas donde se encuentran tales sistemas.

Dado que la codificación y descodificación de vídeo son tareas muy exigentes que requieren una elevada carga computacional, es necesario que el procesador integrado en el sistema embebido proporcione y gestione los recursos *hardware* necesarios para garantizar la realización de esas tareas en tiempo real. Por esta razón, se emplean procesadores cada vez más complejos, con elevadas frecuencias de funcionamiento, lo que deriva a su vez en un incremento del consumo de energía. Este incremento limita severamente la autonomía del dispositivo empleado, siendo necesaria la recarga del mismo continuamente.

La solución clásica para afrontar el problema planteado es reducir el consumo del procesador en base a diseños orientados al bajo consumo que permitan funcionar al dispositivo portable con una menor tensión de alimentación y/o frecuencia. Otra opción para incrementar el tiempo de vida de los dispositivos es afrontar el problema desde el punto de vista del *software* que ejecuta el procesador embebido, analizando el estado de la batería del terminal y actúando en consecuencia, de modo que se ejecuten las aplicaciones con diferentes niveles de consumo. En aplicaciones relacionadas con la presentación de contenidos multimedia, esos niveles de consumo se traducirán en diferentes calidades que apreciará el usuario del terminal.

La problemática planteada, esto es, cómo optimizar el consumo de energía en dispositivos portables multimedia desde el punto de vista de la reducción de la calidad experiencia del usuario a cambio de una mayor autonomía del terminal, se ha convertido en una de las principales líneas de investigación del Grupo de Investigación GDEM (Grupo de Diseño Electrónico y Microelectrónico) de la E.U.I.T. de Telecomunicación perteneciente a la Universidad Politécnica de Madrid.

En concreto, los trabajos desarrollados en esta área, incluyendo el presente Trabajo Fin de Máster, se enmarcan en un proyecto de investigación desarrollado (I+D) por el GDEM denominado PccMUTE (*Power Consumption Control in Multimedia TErminals*). Este proyecto está financiado por el Ministerio de Ciencia e Innovacion (MICINN), con una duración comprendida entre Enero de 2010 y Diciembre de 2012.

Para poder materializar esta propuesta es necesario disponer de un terminal portable basado en algún dispositivo que tenga capacidad de modificar su consumo en función del nivel de carga computacional así como de un reproductor de vídeos con un descodificador de vídeo que soporte la degradación en la calidad de experiencia mencionada.

Como dispositivo portable se trabajará con la plataforma de desarrollo BeagleBoard, un sistema embebido basado en el procesador OMAP3530 que permite modificar la frecuencia de reloj y la tensión de alimentación dinámicamente reduciendo de este modo el consumo del terminal. El procesador OMAP3530 contiene integrado un procesador de propósito general, GPP, (ARM Cortex-A8) y un procesador digital de señal, DSP, (TMS320C64+™).

Por otro lado, como reproductor de vídeos se empleará MPlayer, un reproductor multimedia de libre distribución que soporta la reproducción de vídeos escalables basados en el estándar de codificación de vídeo H.264/SVC (*Scalable Video Coding*). Más concretamente, un video escalable contiene embebidas distintas versiones del video original que pueden ser descodificadas en diferentes resoluciones, tasas de cuadro y calidades (escalabilidades espacial, temporal y de calidad respectivamente), permitiendo una adaptación rápida y muy flexible. De este modo, a través del estándar, se puede degradar la calidad del servicio en función de las necesidades/características del dispositivo.

Este Trabajo Fin de Máster tiene como punto de partida una serie de trabajos previos relacionados con el marco de esta investigación que se presentan a continuación:

- [1] En este trabajo se obtuvo una primera versión funcional del reproductor MPlayer ejecutándose exclusivamente en el procesador de propósito general del OMAP3530.
- [2] En ese trabajo se definió una metodología de optimización en velocidad para descodificadores basados en DSPs, desde el *software* de referencia que implementa el propio descodificador hasta la migración y optimización del mismo, siendo el objetivo principal el funcionamiento en tiempo real.
- [3] En este trabajo se aplicaron las técnicas de optimización definidas en [2] para el descodificador OpenSVC alojado en el procesador digital de señal del procesador OMAP3530, realizando un estudio de las funciones que más recursos empleaban y reduciendo su impacto en el rendimiento del descodificador.

1.2 Objetivos

Por tanto, el principal objetivo de este Trabajo Fin de Máster es la integración del reproductor multimedia Mplayer que se ejecuta en el procesador de propósito general con el descodificador OpenSVC optimizado alojado en el procesador digital de señal con el propósito de distribuir la carga computacional asociada a la descodificación de un vídeo escalable entre los procesadores del OMAP3530.

Este objetivo se desglosa en los siguientes:

- Estudio y análisis en profundidad del reproductor multimedia Mplayer con soporte para el estándar de codificación de vídeo escalable (H.264/SVC) en el núcleo ARM del procesador OMAP3530.
- Estudio del descodificador escalable OpenSVC optimizado para ejecutarse en el núcleo 'C64x+' del procesador OMAP3530.
- Configuración y gestión de memoria para el intercambio de datos entre los procesadores ARM y DSP que integra el OMAP3530.
- Comunicación y sincronización entre los procesadores ARM y DSP que integra el OMAP3530.
- Modificación del código de Mplayer para realizar la descodificación de vídeo escalable en el núcleo 'C64x+'.
- Modificación del código del descodificador OpenSVC para adaptarlo al nuevo método de trabajo realizado en este Trabajo Fin de Máster.
- Comprobación del correcto funcionamiento de la aplicación desarrollada.
- Medidas de conformidad y rendimiento de la aplicación desarrollada.

1.3 Organización de la memoria

La memoria se ha estructurado en seis capítulos. El primer capítulo es esta introducción en la que se enmarca este Trabajo Fin de Máster dentro de la línea de investigación en optimización del consumo de energía en terminales móviles que se lleva a cabo en el GDEM de la Universidad Politécnica de Madrid y se explican sus objetivos.

En el capítulo 2 se presentan las diferentes tecnologías tanto *software* como *hardware* empleadas a lo largo de este trabajo. No se pretende realizar una descripción en profundidad de cada una de ellas, sino cubrir los aspectos generales y destacar aquellos más relevantes que son necesarios para llevar a cabo el presente trabajo. De este modo, se ofrece un resumen del sistema embebido BeagleBoard basado en el procesador OMAP3530, definiendo brevemente los componentes *hardware* por los que está formado así como algunas particularidades desde el punto de vista del sistema operativo embebido que ejecuta el núcleo 'ARM' como la gestión de memoria y la modularidad en sistemas operativos Linux. También se presentan los otros elementos clave de este Trabajo Fin de Máster, como son el reproductor MPlayer, el estándar de codificación de vídeo H.264/SVC, el descodificador OpenSVC y el entorno de desarrollo para aplicaciones 'ARM' y 'C64x+' Code Composer Studio.

En el capítulo 3 se describe la configuración realizada en el sistema completo, esto es, las modificaciones necesarias en las tecnologías presentadas en el capítulo 2 que permiten no sólo la ejecución de aplicaciones en los dos núcleos del procesador OMAP3530, sino también la sincronización y el intercambio de información entre ellos. Para la primera de las tareas, la ejecución de aplicaciones, es necesario adaptar el entorno de desarrollo Code Composer Studio en función del núcleo al que vaya dirigido la aplicación a construir, mientras que para las dos tareas restantes citadas anteriormente, la sincronización y el intercambio de información entre núcleos, se detalla la configuración de los módulos

DSPLink y CMEM que permiten llevar a cabo dichas tareas tareas. Además, a lo largo del capítulo se establece la terminología básica que se empleará posteriormente.

El capítulo 4 cubre el desarrollo principal de este Trabajo Fin de Máster, de manera que a partir de las configuraciones realizadas en el capítulo 3, se describen en profundidad las diferentes etapas necesarias para llevar a cabo la integración entre los dos núcleos del procesador OMAP3530 y la presentación de las imágenes descodificadas en un monitor. Dada la importancia y extensión del capítulo, se proporciona en el inicio del mismo un resumen en el que se definen los puntos a tratar y el orden de su presentación. También se muestran secciones de código comentadas pertenecientes a las funciones que implementan la integración y diagramas de flujo para clarificar las explicaciones realizadas.

El capítulo 5 describe una serie de pruebas parciales realizadas a lo largo de este Trabajo Fin de Máster, las cuales han detectado problemas importantes que han supuesto un gran esfuerzo. Para algunas de estas pruebas se ha sintetizado la información en una serie de puntos con el propósito de facilitar el procedimiento empleado. La segunda parte de este capítulo está dedicada íntegramente a analizar el rendimiento del descodificador OpenSVC a través de una serie de secuencias de vídeo escalable. En relación a los resultados obtenidos, se realizará una comparativa entre diferentes escenarios, todos ellos bajo el sistema embebido BeagleBoard basado en el procesador OMAP3530.

En el capítulo 6 se presentan las conclusiones obtenidas del trabajo realizado y las líneas de trabajo futuro a las que puede dar lugar este Trabajo Fin de Máster.

La memoria concluye con la relación de referencias bibliográficas que se han consultado durante la elaboración de este Trabajo Fin de Máster, junto con una lista de los acrónimos que aparecen en la memoria.

2 TECNOLOGÍAS

En este capítulo se presentan las diferentes tecnologías hardware como software empleadas a lo largo de este Trabajo Fin de Máster y que han sido claves en su desarrollo.

En primer lugar se realiza una breve descripción del sistema embebido BeagleBoard, basado en el procesador OMAP3530, el cual contiene dos núcleos integrados, un procesador de propósito general ARM Cortex A8 y un procesador digital de señal C64x+.

Posteriormente, ligado al sistema operativo embebido en la BeagleBoard, se resumirán las características de la gestión de memoria en Linux así como de los dos módulos del kernel claves empleados: DSPLink y CMEM.

A continuación, se hace referencia a modo de presentación del entorno de desarrollo integrado empleado, Code Composer Studio, el cual ha introducido mejoras significativas respecto a versiones anteriores que han supuesto grandes ventajas para el desarrollo de este trabajo.

Por último, se presentan las dos herramientas software que dan título a este Trabajo Fin de Máster: el reproductor Mplayer y el descodificador escalable OpenSVC. Los aspectos fundamentales de ambas se resumen en este capítulo haciendo mención especial al vídeo escalable, del cual se realiza una breve introducción para tener unas nociones básicas de sus características más relevantes.

2.1 Procesador OMAP3530

El OMAP3530 [4] es un sistema SoC (*System On Chip*) multiprocesador basado en la arquitectura OMAP™3 de Texas Instruments. Los sistemas SOC integran en un único chip toda la funcionalidad de un sistema digital completo basado en microprocesador. Ésto incluye al menos uno o varios núcleos o unidades de proceso (CPUs, *Central Processing Units*), memoria y periféricos. También se pueden encontrar bloques de lógica reconfigurable y bloques analógicos integrados en el mismo chip.

El hecho de integrar todo el sistema, proporciona una gran reducción de tamaño y aumento de la fiabilidad, ya que todos los componentes están sometidos al mismo y único proceso de fabricación, lo que permite reducir costes y tiempo de diseño de los sistemas embebidos finales, como por ejemplo la plataforma de desarrollo empleada en este trabajo, la tarjeta BeagleBoard [5]. Además, la propagación de señales es mejor a nivel de silicio que a nivel de PCB (*Printed Circuit Board*), mejorando así la capacidad de comunicación de los subsistemas internos. El encapsulado que permite esa integración mencionada es de tipo POP (*Package on Package*), ver Fig. 1 , donde la memoria está soldada sobre el propio chip, de modo que no es posible visualizarlo directamente. Con este tipo de encapsulado se consigue reducir el número de pistas necesarias para la conexión de las memorias optimizando el espacio en la PCB.

Fig. 1 Encapsulado del procesador OMAP3530 tipo POP

OMAP3530 es un SoC orientado a aplicaciones multimedia de bajo consumo. Posee dos núcleos o unidades de proceso, un procesador de propósito general ARM Cortex-A8 [6] con capacidad de ejecutar sistemas operativos embebidos genéricos como Linux o Windows y un procesador digital de señal DSP C64x+ (*Digital Signal Processor*) [7], optimizado para trabajar con datos de vídeo en tiempo real. Una de las características más llamativas de OMAP3530 es su bajo consumo, permitiendo ser integrado en sistemas embebidos como la tarjeta BeagleBoard y no superando los 2,5W de potencia en conjunto. Además, incluye 112 KB de memoria ROM y 64 KB de memoria Shared-RAM.

La arquitectura OMAP presenta especialmente un alto rendimiento en procesado de vídeo, imágenes y gráficos. Gracias a ello, este procesador proporciona soporte a alguna de las siguientes funcionalidades:

- *Streaming* de vídeo
- Procesado 2D/3D en dispositivos portátiles.
- Vídeo conferencia.
- Alta resolución de imágenes.
- Captura de vídeo en terminales móviles

La Fig. 2 muestra los diferentes subsistemas que conforman la arquitectura del OMAP3530.

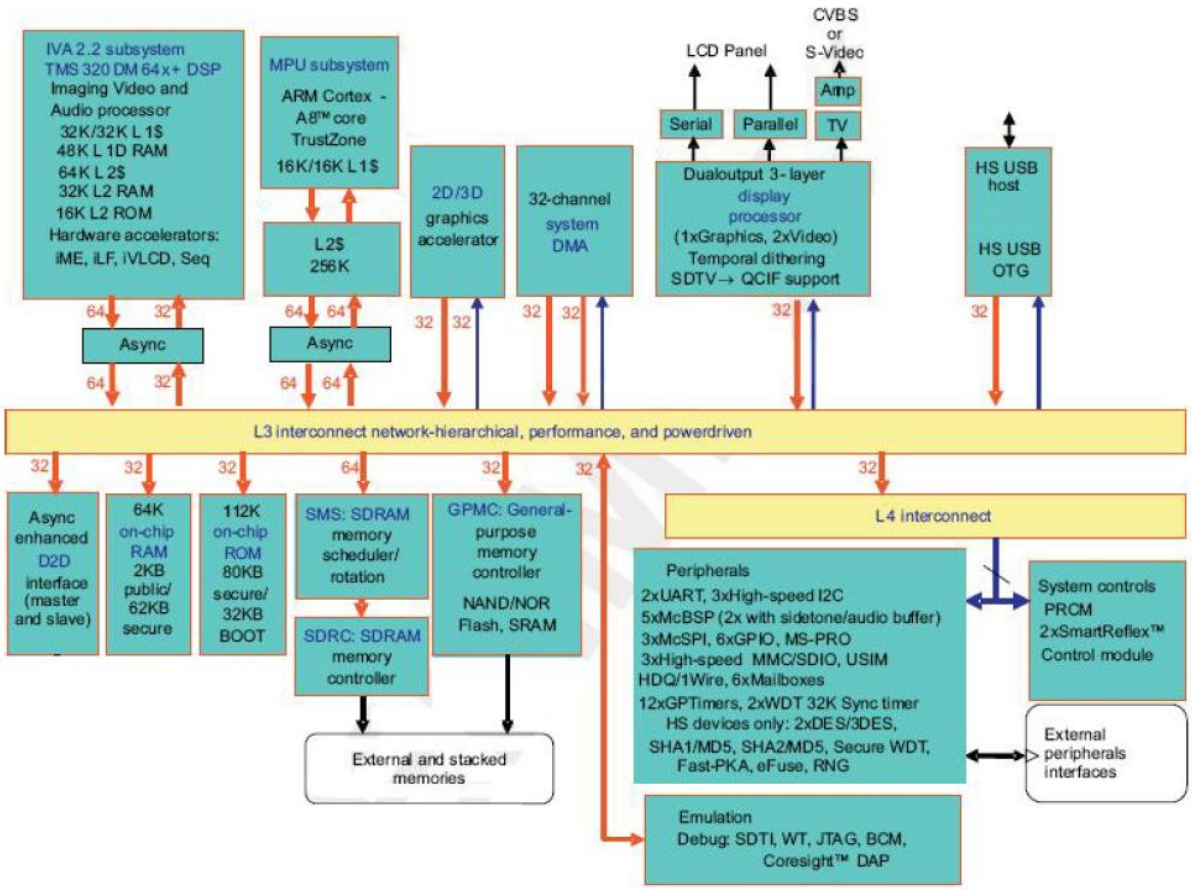

Fig. 2 Arquitectura del procesador OMAP3530

A continuación se describirán los subsistemas principales de la arquitectura del procesador y aquellos que además han sido empleados a lo largo de este trabajo. Todas las descripciones se realizarán desde un punto de vista general, no siendo el objetivo realizar un análisis exhaustivo de cada uno de ellos.

2.1.1 Subsistema MPU - ARM

El elemento principal de este subsistema es el microprocesador ARM Cortex ™ A8, un procesador de propósito general de bajo consumo y de altas prestaciones. Está basado en la arquitectura "superescalar" ARMv7 con trece etapas en el ciclo de instrucción , dos ALUs (*Arithmetic Logic Units*) y una unidad de multiplicación que le permite ejecutar hasta dos instrucciones simultáneamente.

A su vez, integra un coprocesador (NEON) optimizado para aplicaciones multimedia con unidad de multiplicación-acumulación (MAC) propia y soporte para operaciones en coma flotante.

A nivel de memoria, el núcleo ARM contiene dos niveles de memoria caché y su propia MMU (*Memory Management Unit*). El nivel uno está integrado en el propio núcleo y tiene una capacidad de almacenamiento de 16 KB para instrucciones y de 16 KB para datos. El nivel dos tiene una capacidad de 256 KB tanto para instrucciones como para datos. Respecto a la

MMU, ésta es controlada directamente por el sistema operativo para implementar el mecanismo de memoria virtual. Además, la interfaz con el núcleo es de tipo asíncrona, permitiendo un funcionamiento más prioritario.

2.1.2 Subsistema IVA 2.2 – TMS320DM64x+ DSP

El elemento principal de este subsistema es el TMS320C64+, núcleo que incluyen los DSPs de la familia TMS320C6000 de Texas Instruments, junto con el acelerador de vídeo y audio. La funcionalidad del subsistema está claramente definidida para operaciones multimedia (imagen, vídeo y audio).

Es un procesador de 32 bits de punto fijo con arquitectura VLIW (*Very Long Instruction Word*) basado en la versión programable mejorada del núcleo 'C64x'. El núcleo 'C64x+' posee ocho unidades de cálculo con alto grado de independencia en su funcionamiento: dos unidades de multiplicación con operandos de 32 bits y seis ALUs. Dichas unidades soportan instrucciones especializadas para operaciones con datos de vídeo.

En cuanto a su estructura a nivel de memoria, presenta dos niveles que en ambos casos contienen diferentes tipos de memoria. De este modo, el Nivel 1 presenta 32 KB de memoria caché de mapeado directo para programa aceptando configuraciones diferentes (memoria caché/memoria) mediante 32 bytes por línea: 0 KB caché/32 KB memoria, 4 KB/28 KB, 8 KB/24 KB, 16 KB/16 KB, 32 KB/0 KB. Además, este nivel también proporciona 32 KB de memoria caché asociativa (*set associative*) para datos nuevamente configurables y 48 KB de memoria SRAM.

Por otro lado, el segundo nivel de memoria no diferencia secciones entre programa y datos, sino que su tamaño está dividido en tres secciones: 64 KB de memoria caché asociativa por conjuntos configurable mediante 128 bytes por línea, 32 KB de memoria SRAM y 16 KB de memoria ROM.

2.1.2.1 Controlador de DMA del IVA 2.2

Uno de los periféricos más importantes en las aplicaciones de descodificación de vídeo es el controlador de DMA (*Direct Memory Access*), que está integrado en el subsistema IVA 2.2 del OMAP3530. El funcionamiento básico del controlador de DMA es el siguiente: en el subsistema IVA 2.2, la funcionalidad de acceso directo a memoria (DMA, *Direct Memory Access*) se puede llevar a cabo mediante dos mecanismos independientes: utilizando el IDMA (*Internal Direct Memory Access*) o utilizando el EDMA3 (*Enhanced Direct Memory Access v3.0*). La siguiente figura (ver Fig. 3) muestra un diagrama de bloques simplificado en el que se muestra cómo se interconectan los controladores de IDMA y de EDMA3 con el núcleo 'C64x+' y las diferentes memorias.

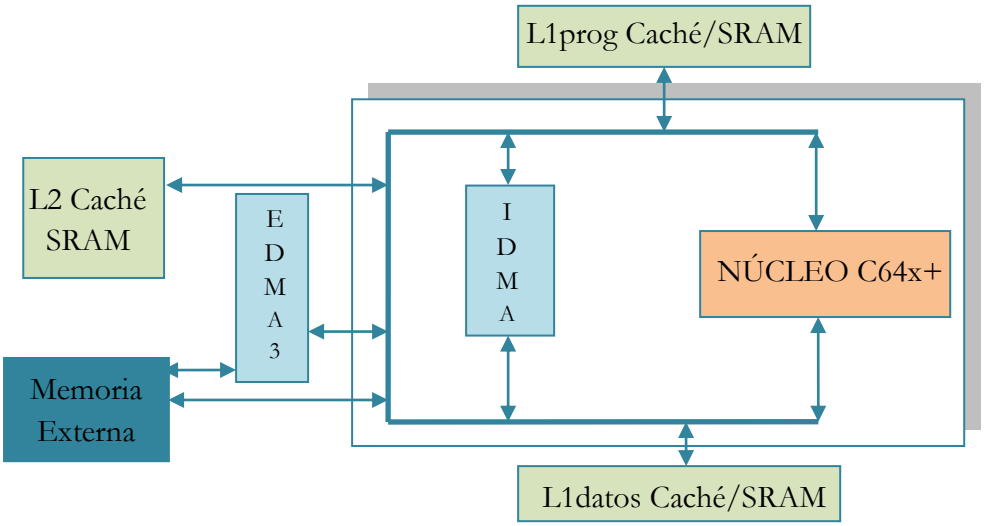

Fig. 3 Controlador de DMA del subsistema TMS320SM64x+ DSP

El controlador IDMA permite realizar transferencias rápidas de datos entre todas las memorias internas y los controladores de periféricos internos con un rendimiento mayor respecto al controlador de caché. El IDMA tiene dos canales: el canal 0 se utiliza para programar de forma rápida periféricos y controladores (por ejemplo, el EDMA3), mientras que el canal 1 se utiliza para transferir datos entre memorias internas (por ejemplo, de L2 a L1 para datos sin intervención del núcleo 'C64x+').

El controlador de EDMA3 se puede utilizar para realizar transferencias de datos entre memoria interna y externa solicitadas por el usuario, o entre diferentes direcciones de memoria externa. A diferencia del controlador de EDMA2 empleado por la familia C6000, el EDMA3 permite realizar transferencias de QDMA encadenadas (*Quick Direct Memory Access*). Se pueden programar y encadenar hasta 128 transferencias diferentes mediante la escritura en los registros de configuración del EDMA3. Una vez que la primera transferencia de QDMA ha sido finalizada, una nueva transferencia comenzará automáticamente. Mediante esta técnica, no hay necesidad de comprobar si cada transferencia de QDMA ha finalizado o no antes de lanzar la siguiente. Además, los datos de configuración de las diferentes transferencias se pueden almacenar en un *buffer* situado en memoria interna L1 para datos, dejando al IDMA la tarea de mover dichos datos a los registros de configuración del EDMA3 sin intervención de la CPU. Esta técnica se lleva a cabo debido a la menor velocidad de escritura en los registros de configuración del EDMA3 respecto a la escritura en memoria interna L1 para datos.

2.1.3 Procesador de vídeo de tres capas (DSS, Display SubSystem)

Es un coprocesador encargado de enviar el vídeo desde la memoria hasta un dispositivo de presentación externo mediante la acción del DMA(*Direct Memory Access*) sin intervención de la CPU. Posee una interfaz digital con pantallas LCD y analógicas mediante conversores de vídeo (transcodificadores de vídeo) internos que proporcionan salida de televisión con formatos NTSC y PAL. Soporta tres canales *hardware* independientes para la gestión de memoria y representación de vídeo: *video1*, *video2* y *graphics*.

En el nivel del *kernel* se encuentran los controladores que permiten manejar estos canales. El controlador V4L2 [8] da soporte a los canales *video1* y *video2* y el driver FRAMEBUFFER [9] al canal *graphics*. Ambos controladores permiten la reserva, configuración de memoria y envío de vídeo a los dispositivos externos ocultando al usuario las operaciones complejas de configuración de los periféricos.

2.1.4 Controladores de memoria externa

El controlador de memoria de propósito general, GPMC (*General Propose Memory Controller*), puede interactuar con memorias RAM estáticas y memorias no volátiles tipo FLASH. También puede interactuar con periféricos externos que tengan capacidad de almacenamiento interno como, por ejemplo, un chip *Ethernet*.

El controlador de memorias síncronas SDRC (*SDRam Controller*) puede interactuar con memorias RAM síncronas, típicamente memorias tipo DDR (*Double Data Rate*), que normalmente se utilizan como memoria principal del sistema operativo.

2.1.5 Gestión de consumo de energía (PRCM, *Power Reset and Clock Management*)

El OMAP3530 incorpora tecnología específica para el control y la optimización de consumo de energía. El chip está dividido en nueve dominios de potencia, cada uno de ellos asociado a una red de alimentación independiente con capacidad de escalado del nivel de tensión. También es posible la conexión y la desconexión de cada dominio permitiendo optimizar el consumo en función de las necesidades de la aplicación.

2.1.6 Periféricos

El OMAP3530 cuenta con numerosos periféricos de comunicaciones y buses estándares que le otorgan una gran capacidad de interconexión con sistemas externos dotando al procesador de unos niveles de flexibilidad y adaptabilidad muy elevados. Algunos de ellos son: tres interfaces I2C de alta velocidad para comunicaciones maestro-esclavo, tres interfaces de comunicaciones serie asíncronas, controlador USB2.0 OTG de alta velocidad, doce temporizadores de propósito general, controlador *host* multipuerto de alta velocidad, interfaz de comunicación maestro-esclavo para dispositivos SPI, etc. Todos los periféricos están conectados al resto de módulos del sistema a través del nivel de interconexión cuatro, excepto el módulo USB que lo hace directamente a través del bus de nivel tres. Ambos niveles de interconexión proporcionan un alto ancho de banda para transferencia de datos entre los periféricos del chip, la memoria externa y la interna.

2.2 Sistema embebido BeagleBoard – Perspectiva Hardware

La tarjeta BeagleBoard es un sistema embebido basado en el procesador OMAP3530 diseñado y comercializado por la empresa DigiKey [10]. El origen del nombre es un acrónimo de las características de esta tarjeta, las cuales pueden verse en la siguiente figura (ver Fig. 4):

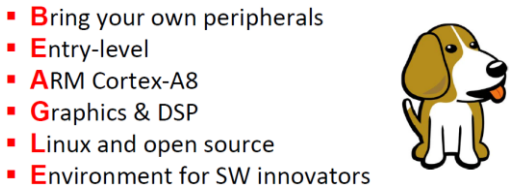

Fig. 4 Presentación de características (acrónimo) del sistema embebido BeagleBoard

La tarjeta BeagleBoard, de bajo coste, es ampliamente usada en el área de los sistemas embebidos a nivel mundial y contiene numerosas funcionalidades dado el gran número de periféricos que posee, lo que le hace ser muy interesante para el desarrollo de aplicaciones.

La empresa que se encarga de su comercialización, Digikey, ofrece un portal web "www.BeagleBoard.org" como soporte para el desarrollo de aplicaciones en la tarjeta. Además, se puede encontrar documentación sobre la tarjeta, notas de aplicación, proyectos realizados con la tarjeta como elemento principal, un foro donde poder intercambiar opiniones, dudas y comentarios con otros usuarios, etc.

Para la realización de este Trabajo Fin de Máster se ha empleado la versión C4 (ver Fig. 5), la cual incorpora respecto a versiones anteriores mayor capacidad de memoria pasando de 128 MB a 256 MB. BeagleBoard ha seguido evolucionando implementando nuevos sistemas embebidos como BeagleBone y BeagleBoard-xM, el cual presenta un SoC que incorpora dos núcleos de propósito general pero ningún procesador digital de señal, de manera que la tarjeta empleada, BeagleBoard, es la más adecuada al trabajo realizado.

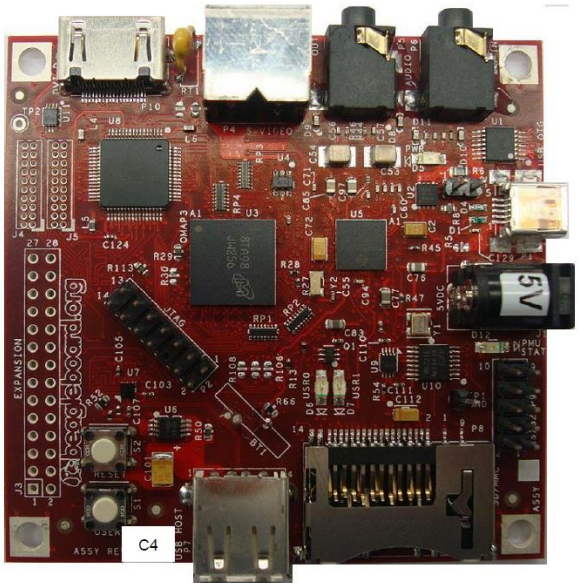

Fig. 5 Sistema embebido BeagleBoard – Versión C4

Como se ha comentado anteriormente, la tarjeta BeagleBoard está basada en el procesador OMAP3530, pero no todas las funcionalidades del procesador están disponibles a través de la tarjeta sino que ésta contiene un conjunto de características reducido pero suficiente para abordar las tareas realizadas en este trabajo. De hecho, el manual de la tarjeta BeagleBoard proporciona información sobre las interfaces y funcionalidades que sí están accesibles en la propia tarjeta. Son las siguientes:

- Subsistema MPU basado en el microprocesador ARM Cortex-A8™.
- Interfaz de memoria POP
  - 2 GB de memoria DDR (256 MB)
  - 2 GB de memoria Flash NAND (256 MB)
- Interfaz de display RGB de 24 bits (DSS)
- Interfaz SD/MMC
- Interfaz USB OTG
- Salidas NTSC/PAL/S-Video
- Gestión de consumo de energía
- Interfaz serie
- Interfaz I2C
- Interfaz de audio (McBSP2)
- Expansión McBSP1
- Interfaz de depuración JTAG

Esa reducción de características en la tarjeta no significa que las aplicaciones que se puedan llevar a cabo en la tarjeta carezcan de complejidad, sino que toda la funcionalidad del procesador OMAP3530 no se puede implementar en la tarjeta BeagleBoard debido a su bajo coste de diseño y fabricación. Pese a ello, los recursos disponibles proporcionan la suficiente funcionalidad como para desarrollar aplicaciones de alta complejidad como la realizada en este proyecto.

El manual de referencia de la tarjeta [11] proporciona algunos escenarios donde se puede emplear la tarjeta BeagleBoard como parte del sistema (ver Fig. 6). Por ejemplo, la tarjeta puede ser "convertida" en una unidad central de procesamiento (CPU) añadiendo las interfaces necesarias (teclado y ratón) y elementos de audio e imagen para poder escuchar y visualizar los contenidos. Otro modo de trabajo puede hacer que la tarjeta funcione como una interfaz para la visualización de imágenes provenientes de equipos externos a la BeagleBoard, etc.

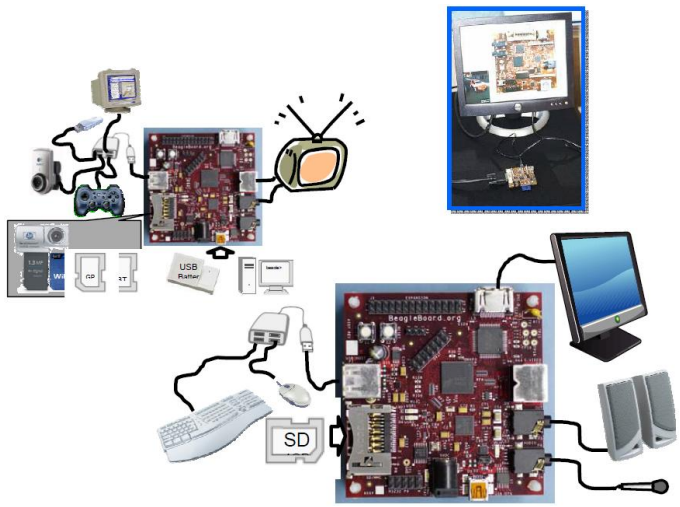

Fig. 6 Escenarios de aplicación del sistema embebido BeagleBoard

Precisamente, el primero de los escenarios descritos es el que se ha implementado en el laboratorio donde se ha llevado a cabo este trabajo y que será descrito con más detalle en capítulos posteriores.

En cuanto a la arquitectura de la tarjeta, la Fig. 7 representa un diagrama de bloques de ésta en la cual puede observarse como el procesador OMAP3530 de Texas Instruments es el elemento principal de su arquitectura. A la derecha de éste se puede observar el integrado de alimentación, TPS65950, que se encarga, entre otras tareas, de gestionar la interfaz de comunicaciones de algunos periféricos. Dado el elevado número de ellos, la complejidad del integrado para gestionar las tensiones de alimentación y su consumo es elevada. Por otro lado, el OMAP3530 incorpora los recursos necesarios para controlar algunos periféricos, tales como el pulsador de *reset*, la tarjeta SD/MMC, los pines de expansión, el conector JTAG y la salida analógica de vídeo S-Vídeo. Sin embargo, otros periféricos como la salida DVI-D necesitan controladores específicos como el chip TFP410, ya que el OMAP3530 no puede controlarlos directamente.

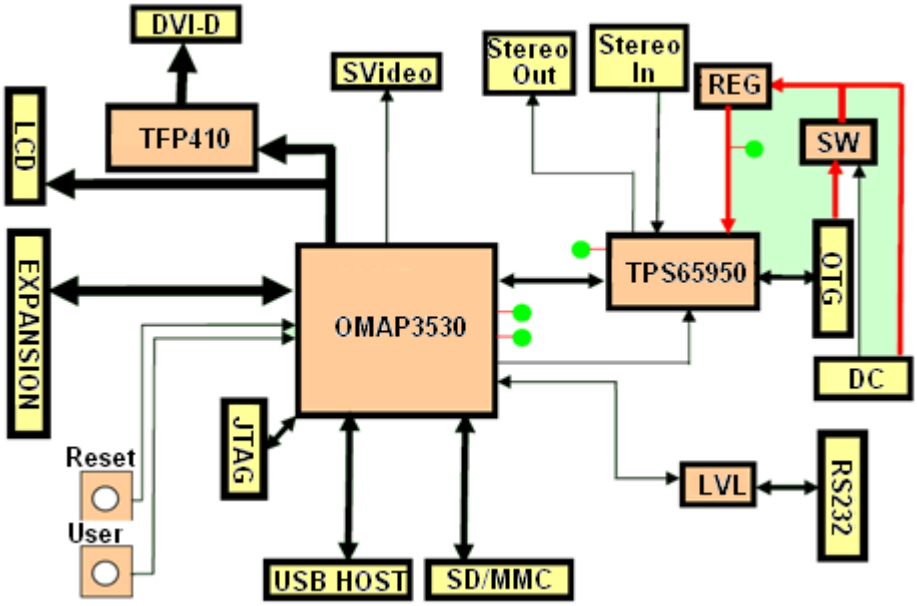

Fig. 7 Diagrama de bloques del sistema embebido BeagleBoard

El procesador OMAP3530, núcleo de la arquitectura de la tarjeta, ya se analizó en el apartado anterior, de forma que se procederá a describir los otros subsistemas de mayor importancia y que han sido usados a lo largo de este trabajo.

2.2.1 Memoria

La tarjeta BeagleBoard lleva instalada una memoria "Micron POP" soldada sobre el chip de silicio OMAP3530 que presenta las siguientes características:

- 2 GB de memoria MDDR SDRAM con bus de 32 líneas (256 MB).
- 2 GB de memoria NAND con bus de 16 líneas (256 MB).

Es posible añadir más memoria instalando una memoria NAND a través del puerto SD/MMC o el puerto USB OTG.

2.2.2 Control de alimentación TPS65950

Este subsistema [12] proporciona soporte principalmente al sistema de alimentación y reinicio de la tarjeta. El TPS65950 alimenta tanto al procesador OMAP3530 como a la tarjeta mediante tres tensiones, siendo ajustable la primera de ellas. Esta tensión ajustable es la del procesador y presenta como valor inicial 1.2 V en el arranque, pero posteriormente puede ser ajustada por *software* hasta 1.3 V. Las otras dos tensiones de alimentación son fijas y presentan valores de 1.3 V y 1.8V. Dada la amplia funcionalidad del integrado TPS65950, éste es controlado desde el OMAP3530 mediante la interfaz I^2C. Además de los voltajes mencionados existen otros seis voltajes más generados por el TPS65950 encargados de alimentar a algunos de los periféricos integrados en el OMAP3530 como el PLL y su circuitería o el *hardware* para la SD/MMC.

2.2.3 Conector DC

Empleado para alimentar la tarjeta BeagleBoard. La tensión de entrada debe ser de 5 V y la corriente debe ser como mínimo de 50 mA y en ningún caso superar los 2 A.

2.2.4 Puerto USB 2.0 OTG

Este puerto es la primera fuente de comunicación y alimentación para la BeagleBoard. A través de un cable USB conectado a un ordenador, se puede alimentar a la BeagleBoard trabajando del mismo modo que alimentando a la tarjeta a través del conector DC . En este Trabajo Fin de Máster se ha usado este puerto para conectar un adaptador USB-Ethernet, que ha formado parte de la configuración inicial para ejecutar un sistema de ficheros a través del protocolo de red NFS (*Network File System*).

Obviamente, cuando no se usa el conector USB OTG para alimentar la tarjeta, es necesario alimentar ésta a través del conector de alimentación DC.

2.2.5 Conector JTAG

Es un conector de 14 pines que incorpora la tarjeta BeagleBoard para permitir el desarrollo y depuración de *software* en el procesador OMAP3530. En concreto, para el desarrollo de este trabajo se ha empleado el JTAG para depurar el *software* que se ejecuta en el núcleo 'C64x+' del OMAP3530 a través un Entorno de Desarrollo Integrado que será presentado más adelante.

2.2.6 Conector serie RS-232

La tarjeta BeagleBoard incorpora un conector de 10 pines para permitir comunicaciones serie RS-232. Este conector está comunicado con la UART3 del procesador OMAP3530, siendo necesario incorporar entre ambos un dispositivo para adaptar los niveles de tensión de la comunicación RS-232 a los de la UART3 empleada para manejar la comunicación serie (CMOS). A su vez, para poder conectar el puerto serie de la tarjeta BeagleBoard con el puerto serie de un PC cualquiera, se requiere un cable IDC a DB9.

La conexión a través de RS-232 es empleada como interfaz de usuario para mostrar un terminal de consola del sistema operativo embebido en el OMAP3530[1], permitiendo el envío y recepción de comandos. Esta terminal ha sido ampliamente usada en el arranque del sistema operativo y durante la ejecución de aplicaciones en el OMAP3530.

2.2.7 Conector MMC/SD

Este conector es uno de los principales puertos de expansión de la tarjeta BeagleBoard. Prueba de ello es el amplio número de dispositivos y tecnologías que acepta. Algunos de ellos se muestran a continuación:

- Tarjetas Wi-Fi
- Cámara
- Tarjetas Bluetooth
- Módulos GPS
- Tarjetas de memoria SD
- Tarjetas de memoria MMC

En la parte de configuración del entorno de trabajo, la principal función que ha tenido el conector MMC/SD ha sido la de poder insertar tarjetas de memoria SD, las cuales contenían los ficheros necesarios para poder realizar el arranque de sistema operativo en el procesador OMAP3530.

2.2.8 Conector DVI-D

La tarjeta BeagleBoard se puede conectar a un monitor LCD que lleve incorporado una entrada DVI-D. Para ello, el procesador OMAP3530 ofrece un interfaz DVI-D que

[1] En concreto, el sistema embebido se ejecuta en el microprocesador de propósito general ARM Cortex – A8. Este tema se describirá más adelante.

soporta 24 bits de color de salida. Dado el gran tamaño del conector DVI-D y a la falta de espacio en la tarjeta, ésta incorpora un conector HDMI, de dimensiones reducidas, por lo que para conectar un monitor con entrada DVI-D se debe usar un cable con un conector DVI-D en un extremo (monitor) y un conector HDMI en el otro extremo (BeagleBoard).

2.2.9 Conector de audio entrada/salida

La tarjeta incorpora una salida y una entrada de audio estéreo con conectores jack de 3,5 mm. Esta salida y entrada de audio permite grabar y reproducir sonido simultáneamente.

2.2.10 Pines de expansión

Se dispone de 28 pines adicionales para conectar varias tarjetas de expansión y ampliar la funcionalidad de la tarjeta.

Para terminar, la siguiente figura (ver Fig. 8) muestra nuevamente la capa superior de la tarjeta BeagleBoard, pero en esta ocasión se han sobrescrito en la imagen los diferentes subsistemas y conectores descritos para tener una perspectiva visual de cada uno de ellos, conociendo su posición en la tarjeta.

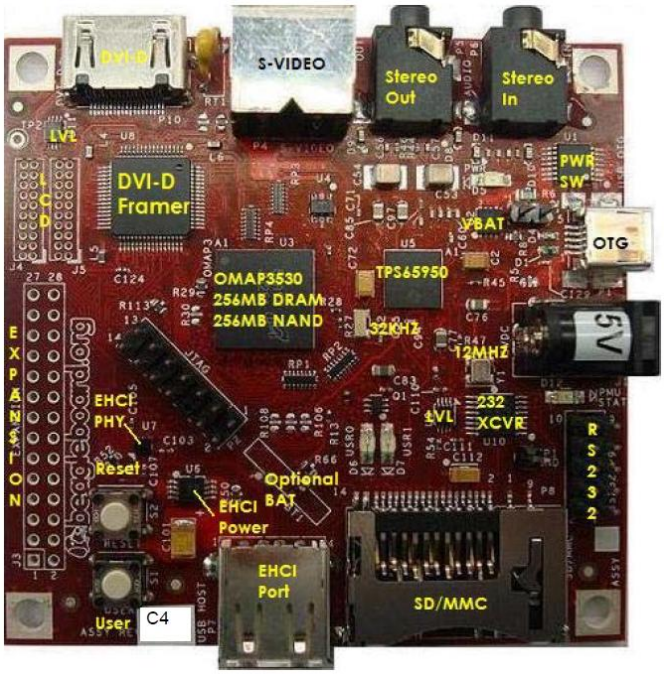

Fig. 8 Ubicación de elementos *hardware* sobre el sistema embebido BeagleBoard

2.3 Sistema embebido BeagleBoard – Perspectiva Software

Una vez vistos los diferentes componentes e interfaces *hardware* que componen el sistema embebido BeagleBoard basado en el procesador OMAP3530 y su funcionalidad dentro del mismo, se va a realizar una descripción desde una perspectiva *software* de la arquitectura de la tarjeta, destacando los principales elementos que la conforman.

Como ya se dijo en el apartado *hardware* de la BeagleBoard, el procesador OMAP3530 contiene un núcleo 'ARM' y un núcleo 'C64x+', por lo que procesos como la comunicación o la sincronización entre núcleos deben poder llevarse a cabo para explotar el potencial de la tarjeta, ya que un núcleo estará más optimizado para unas tareas mientras que el otro lo estará para otras. Por tanto, debe existir un elemento coordinador con capacidad de planificar y gestionar los recursos *hardware* y proporcionar servicios a los programas que así lo requieran, en definitiva, un sistema operativo. Dada la arquitectura del núcleo 'ARM', éste ofrece la posibilidad de ejecutar un sistema operativo embebido con un grado de flexibilidad elevado en cuanto a la variedad del mismo: Windows CE, Symbian, Linux, Android, etc. En el Grupo de Investigación GDEM se decidió trabajar con un sistema operativo Linux[2], dada la compatibilidad existente con el procesador OMAP3530 y las ventajas que supone el empleo de un sistema operativo modular como Linux, característica que permite adaptar el sistema a una arquitectura determinada empleando sólo los elementos realmente necesarios, lo que eleva enormemente la eficiencia del sistema.

Existen ciertos requisitos mínimos *hardware* que debe poseer un procesador de propósito general ARM para que sea capaz de iniciar un sistema operativo Linux. En primer lugar, se necesita una CPU de 32 bits, y que posea un gestor *hardware* que controle los accesos a memoria de la CPU, es decir, una MMU (*Memory Management Unit*)[3]. Además, el procesador debe tener una cantidad suficiente de RAM que sea capaz de albergar el propio sistema operativo. El núcleo 'ARM' que posee el procesador OMAP3530 cumple con estos requisitos.

Conocidas y cumplidas las características mínimas, la siguiente figura (ver Fig. 9) muestra la estructura de un sistema operativo Linux, la cual se describirá someramente a continuación.

[2] En realidad, el sistema operativo completo es "GNU/Linux", que está compuesto por el núcleo Linux y un conjunto esencial de aplicaciones GNU que completan el sistema operativo.

[3] Existen versiones especialmente modificadas de Linux (uClinux) [13] que son capaces de ejecutarse en CPUs que no posean MMU.

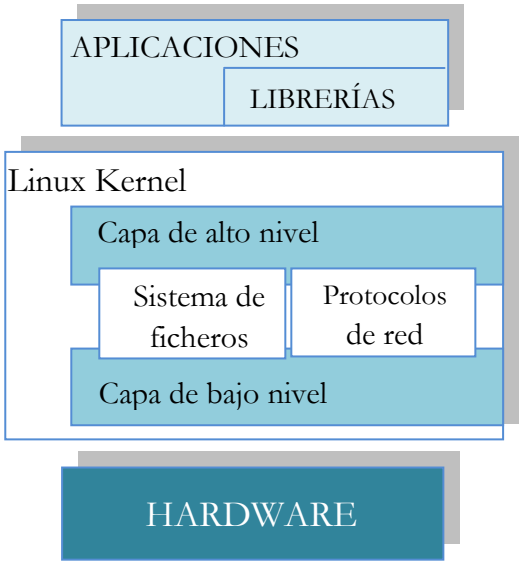

Fig. 9 Estructura de un sistema operativo Linux

Como puede verse en la Fig. 9, inmediatamente después del *hardware* se encuentra el núcleo (*kernel*) del sistema. El *kernel* es el componente básico del sistema operativo y su principal función es la de gestionar el *hardware* de una manera coherente proporcionando una abstracción de alto nivel para el *software* del nivel de usuario. Dentro de esa gestión coherente se debe manejar los distintos dispositivos del sistema, controlar las operaciones de entrada/salida, manejar la planificación de procesos, controlar el espacio de memoria, etc. Las aplicaciones que empleen el API suministrada por el *kernel* Linux podrán ser portadas a todas las arquitecturas que soporten ese *kernel*, con cambios mínimos o sin cambios. Dentro del *kernel* pueden diferenciarse distintas capas:

- Capa de bajo nivel (*Low-level interfaces*): encargada de manejar el *hardware* directamente empleando el API específico de cada arquitectura.
- Capa de alto nivel (*High-level abstractions*): proporciona una abstracción común para todos los sistemas UNIX, permitiendo el control de ficheros, *sockets* y procesos.
- Capas intermedias: suelen contener elementos de interpretación que adapten las estructuras de datos para/desde determinados dispositivos. Los sistemas de ficheros o los protocolos de red son ejemplos de capas intermedias.

Mencionar que la capa de alto nivel es generalmente similar para todas las arquitecturas, ya que la abstracción del *hardware* se consiguió en la capa de bajo nivel.

El otro elemento principal que compone el sistema operativo es el sistema de ficheros. Es más, el *kernel* necesitará al menos un sistema de ficheros correctamente estructurado para funcionar. Será desde este sistema de ficheros donde el *kernel* cargará la primera aplicación que deba ejecutarse en el sistema así como muchas otras operaciones tras realizarse la carga completa del sistema como por ejemplo la inserción de nuevos módulos del sistema o proporcionar un directorio de trabajo a cada proceso del sistema. El sistema de ficheros puede encontrarse puede encontrarse físicamente en un dispositivo de almacenamiento (como un disco duro o una tarjeta de memoria SD/MMC), o puede cargarse en memoria RAM durante el inicio del sistema y operar desde ahí.

Particularizando para este Trabajo Fin de Máster, entre todos los sistemas de ficheros disponibles (con diferentes gestores de ventanas y un número moderado de herramientas instaladas), se ha elegido el correspondiente a la distribución de Angström [14] con Xfce como gestor de ventanas y un nivel medio de herramientas integradas[4]. El hecho de seleccionar Xfce para la gestión de ventanas se debe al reducido consumo de recursos del sistema (tiempo de CPU y memoria RAM) y al elevado grado de madurez, estabilidad y usabilidad de éste, características idóneas para su uso en sistemas embebidos. Respecto a la distribución Angström, ésta comenzó a través de pequeños grupos de trabajo involucrados en diferentes proyectos (OpenEmbedded, OpenZaurus y OpenSimpad) con el fin de unir sus fuerzas y crear una distribución estable, atractiva para el usuario y dirigida a sistemas embebidos. El objetivo se materializó y, como se ha podido comprobar en el trabajo realizado, la distribución ha funcionado de forma eficiente.

Un último aspecto importante a resaltar es la ubicación y el acceso al sistema de ficheros embebido en el núcleo 'ARM'. Normalmente, las diferentes distribuciones se encuentran almacenadas en el propio sistema embebido de distintas formas: en una memoria FLASH, una tarjeta de memoria SD, etc. pero Linux (a nivel de *host*) permite que también se encuentre en un sistema remoto al que se esté conectado por red. La primera de las opciones planteadas permite hacer el sistema portable, pero tiene la desventaja de que las modificaciones que se quieran hacer en ficheros del propio sistema de ficheros deben hacerse desde el propio dispositivo portátil o apagando el dispositivo, sacando la tarjeta de memoria SD, introducirla en el *host* de desarrollo, realizar las modificaciones deseadas, volverla a introducirla ahora en el dispositivo portátil y reiniciar éste, lo que se convierte en un proceso incómodo, lento y muy poco eficiente para entornos de desarrollo. Sin embargo, la segunda opción (el acceso a través de la red) permite introducir cambios en el sistema de ficheros de la tarjeta de forma rápida y cómoda, pues simplemente supondrá acceder a un directorio que se encuentra en el disco duro del *host* de desarrollo compartido por red. Para ello, se hará uso del protocolo de nivel de aplicación NFS (*Network File System*) [15], el cual proporciona soporte a distintos sistemas conectados a una misma red para que accedan a ficheros remotos como si se trataran de locales. Por último, será necesario indicar al *kernel* del Linux embebido la ruta del sistema de ficheros compartido a través de NFS. La Fig. 10 muestra el arranque de la distribución Angström en la tarjeta BeagleBoard a través del monitor.

Fig. 10 Arranque de la distribución Angström ejecutándose en el núcleo 'ARM'

[4] Por nivel medio de herramientas se entiende que se encuentran incluidas las herramientas básicas y todas las herramientas específicas que han sido necesarias para el trabajo realizado.

A continuación se van a abordar, desde una perspectiva genérica, dos temas de especial importancia en el desarrollo del trabajo realizado que están interconectados entre sí y que son necesarios para comprender el funcionamiento de la aplicación realizada. Son el manejo de memoria en Linux y la inserción y empleo de módulos en el *kernel* Linux embebido. La alta dependencia entre ambos temas se debe al empleo de módulos cuya funcionalidad está directamente relacionada con la gestión de memoria que realiza el sistema operativo Linux que se ejecuta en el núcleo 'ARM' del procesador OMAP3530. Por tanto, en primer lugar se realizará una descripción del manejo de memoria en Linux, comentado sus particularidades; después se hablará sobre la modularidad del sistema operativo Linux, y para finalizar, se dará una breve introducción a dos de los principales módulos que han sido ampliamente usados a lo largo de este trabajo: DSP/BIOS Link [16] y CMEM (*Contiguous Memory Allocator*) [17]. La configuración, funcionalidad y su estrecha relación con temas de memoria serán analizadas en detalle en los capítulos 3 y 4.

2.3.1 Gestión de Memoria en Sistemas Operativos Linux

El principal concepto que introduce el sistema operativo Linux es el de la diferenciación entre espacio de memoria virtual y espacio de memoria real. Esta distinción entre ambos espacios realizada por el Linux embebido que se ejecuta en el núcleo 'ARM' dificulta el diseño de aplicaciones conjuntas con el 'C64x+', puesto que ambos núcleos controlan espacios de memoria distintos. El espacio de memoria real del sistema operativo es en el que se encuentran todos los periféricos del sistema y todas las direcciones de memoria. Esta zona se empleará para el desarrollo de *drivers*, y en el caso particular de este trabajo, para realizar lecturas y escrituras en secciones de memoria que sean accesibles por el núcleo 'C64x+'. También se realizarán en esta zona las operaciones que involucran a los cargadores de arranque primario y secundario así como el código que se ejecuta en memoria ROM interna de la tarjeta. Sin embargo, desde el momento en el que se inicia el *kernel* Linux, los programas emplearán el espacio de memoria virtual.

Este espacio de memoria virtual se dividirá a su vez en dos secciones, el espacio de memoria del *kernel* y el espacio de memoria de usuario (ver Fig. 11).

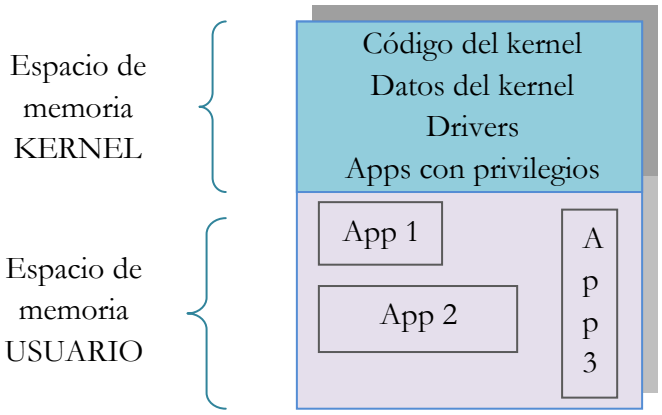

Fig. 11 Gestión de memoria en sistemas operativos Linux

El espacio de memoria de usuario es aquel dónde se encuentran las aplicaciones, mientras que el espacio de memoria del *kernel* es el espacio de memoria donde están los procesos del núcleo del sistema operativo. Esta división se realiza con el fin de no permitir que errores en aplicaciones de usuario corrompan procesos esenciales del sistema operativo. El *kernel* y las aplicaciones que se ejecutan desde el espacio de memoria del *kernel* pueden acceder a cualquier dirección de memoria sin restricciones. Sin embargo, cada aplicación dentro del espacio de memoria de usuario tendrá un espacio de memoria distinto, el cual le será otorgado por el *kernel* al inicio de dicha aplicación, y una aplicación no podrá acceder al espacio de memoria de otra aplicación, tal y como muestra la Fig. 11.

La traducción entre el espacio de memoria virtual y el espacio de memoria real se realiza a través de tablas de direcciones, lo que permite una gran flexibilidad a la hora de gestionar las diferentes regiones de acceso a memoria. De este modo, será posible incluir en el espacio de memoria de un proceso (aplicación) determinados datos, ya que éstos podrán ser enlazados directamente al espacio de memoria virtual de dicho proceso. Sin embargo, no todo el espacio de memoria virtual tiene una traducción en direcciones de memoria reales del sistema. Es más, pueden existir zonas del espacio de memoria virtual de un proceso que no se empleen de forma habitual durante la ejecución del sistema, por lo que estas zonas se eliminarán de la memoria real. Cuando el proceso necesite de nuevo esas zonas de memoria, se volverán a cargar en memoria real y se actualizará la tabla de direcciones del proceso. Esto permite al *kernel* asignar más memoria a los procesos de la que realmente se dispone en el sistema.

La Fig. 12 muestra un ejemplo acerca de la estructuración de la memoria con los procesos X e Y.

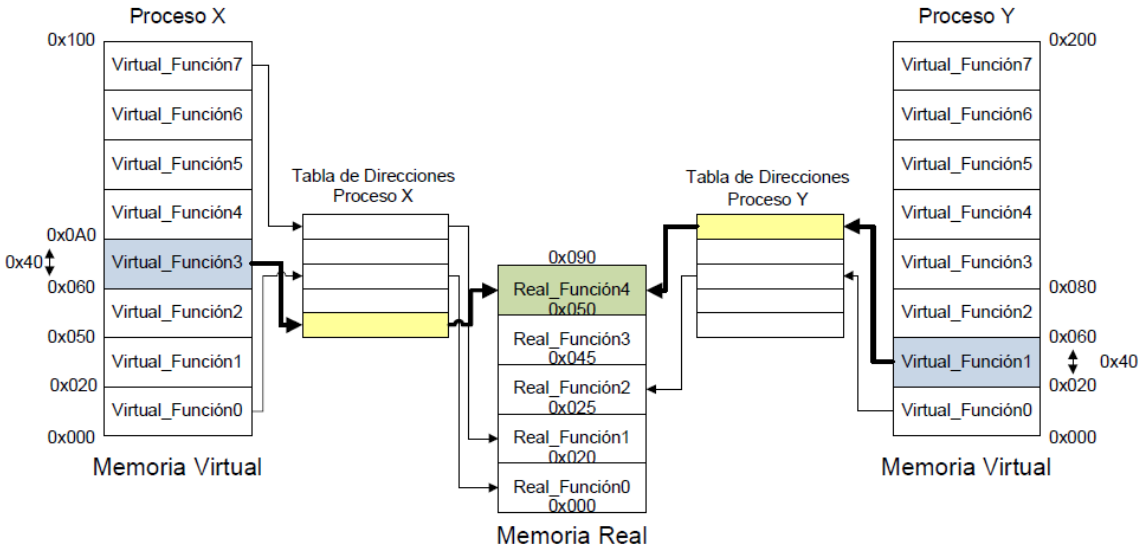

Fig. 12 Estructuración de la memoria mediante dos procesos

En ella se muestra cómo los procesos X e Y emplean la misma función pero únicamente el sistema operativo la almacena una única vez. Para que ambos procesos puedan emplearla, se mapea la función dentro del espacio de memoria virtual de cada proceso a través de la tabla de direcciones de cada uno de ellos. Es importante destacar que la misma dirección de memoria virtual para distintos procesos puede traducirse en direcciones de memoria reales

diferentes. Así pues, siguiendo con el ejemplo de la Fig. 12, las direcciones de memoria virtuales comprendidas entre 0x000 y 0x020 de los procesos X e Y se traducen en direcciones de memoria reales diferentes: en el caso del proceso X entre las direcciones reales 0x000 y 0x020 y en el caso del proceso Y entre las direcciones 0x025 y 0x045.

En definitiva, la gestión de memoria de Linux implica la necesidad de emplear herramientas adicionales para poder comunicar los dos núcleos del procesador OMAP3530, requisito fundamental para alcanzar el objetivo de este Trabajo Fin de Máster. Esas herramientas adicionales serán módulos del *kernel* y sus funcionalidades permitirán "superar" los impedimentos puestos por el sistema operativo Linux y su gestión de la memoria.

2.3.2 Modularidad en Sistemas Operativos Linux

El *kernel* Linux es un núcleo de sistema operativo "monolítico hibrido", esto es, monolítico porque es un gran conjunto de componentes en el que todos operan dentro del mismo espacio y todos ellos tienen acceso a todas las estructuras de datos y rutinas del núcleo, e hibrido porque es posible cargar y descargar nuevos módulos del *kernel* incluso en tiempo de ejecución.

Los módulos del *kernel* son partes de código que pueden ser enlazadas dinámicamente con el núcleo del sistema incluso después del inicio del mismo. También pueden ser desenlazados y borrados cuando ya no se necesiten. La mayor parte de los módulos del *kernel* Linux son *drivers* de dispositivos o *pseudo-drivers* como manejadores de sistemas de ficheros. En el caso de este trabajo, los módulos del *kernel* que se han insertado ofrecen soporte para sincronizar y comunicar, entre otras tareas, los núcleos 'ARM' y 'C64x+' del procesador OMAP3530, de tal forma que cuando se carga un módulo del *kernel*, éste se convierte en una parte más del código del *kernel*, pasando a tener sus mismos derechos y capacidades. Más adelante (ver apartado 3.2.2.3) se detallarán los métodos seguidos para realizar la inserción de los diferentes módulos en el *kernel* y la diferencia entre cargar en tiempo de arranque y ejecución del sistema operativo.

Sí es importante resaltar la dependencia del módulo con respecto a la versión del *kernel* sobre el que se pretende insertar. Generalmente, cuando el usuario se decide por una herramienta *software*, el fabricante proporciona el código fuente de la misma y ficheros *makefile* que contienen las órdenes para generar el módulo. Una de esas órdenes hace referencia al código fuente perteneciente al *kernel* que está ejecutándose en el sistema embebido, lo que obliga al usuario a compilar los módulos del *kernel* para la versión exacta del *kernel* empleado en la tarjeta ya que de lo contrario no será posible emplear las funcionalidades de éste.

A continuación se presentarán los dos módulos empleados a lo largo de este Trabajo Fin de Máster que han tenido una importancia significativa en el desarrollo del mismo y que a su vez han sido fuente de problemas constantes. El primero de ellos, DSP/BIOS Link ha sido empleado principalmente para la comunicación/sincronización entre núcleos de la tarjeta BeagleBoard y el intercambio de información de tamaño reducido entre ellos. El segundo módulo, CMEM, ha servido para intercambiar grandes bloques de información entre

núcleos, solucionando la gestión de la memoria por parte del sistema operativo Linux en el núcleo 'ARM' que se comentará a continuación.

2.3.2.1 DSP/BIOS LINK

DSP/BIOS Link es la herramienta *software* facilitada por Texas Instruments para la comunicación entre los dos núcleos del sistema embebido BeagleBoard. Esta herramienta proporciona una API genérica que permite abstraerse de las características de la capa física conectando dos arquitecturas ARM y DSP desde el nivel de aplicación. DSPLink se puede emplear tanto en sistemas SoC, donde ambos procesadores se encuentran integrados en un mismo *chip* de silicio, como en sistemas multiprocesador discretos, en los cuales los dos procesadores se encuentran separados físicamente. Los requisitos de DSPLink se adaptan perfectamente al *hardware* empleado en este trabajo por lo que se convierte en la herramienta idónea para las tareas requeridas de:

- Sincronización y comunicación entre ambos núcleos del procesador OMAP3530.
- Transferencia de información reducida a través de colas de mensajes.

La arquitectura *software* de DSPLink se presenta a continuación (ver Fig. 13) en la cual se establecen diferencias estructurales para arquitecturas ARM y DSP genéricas.

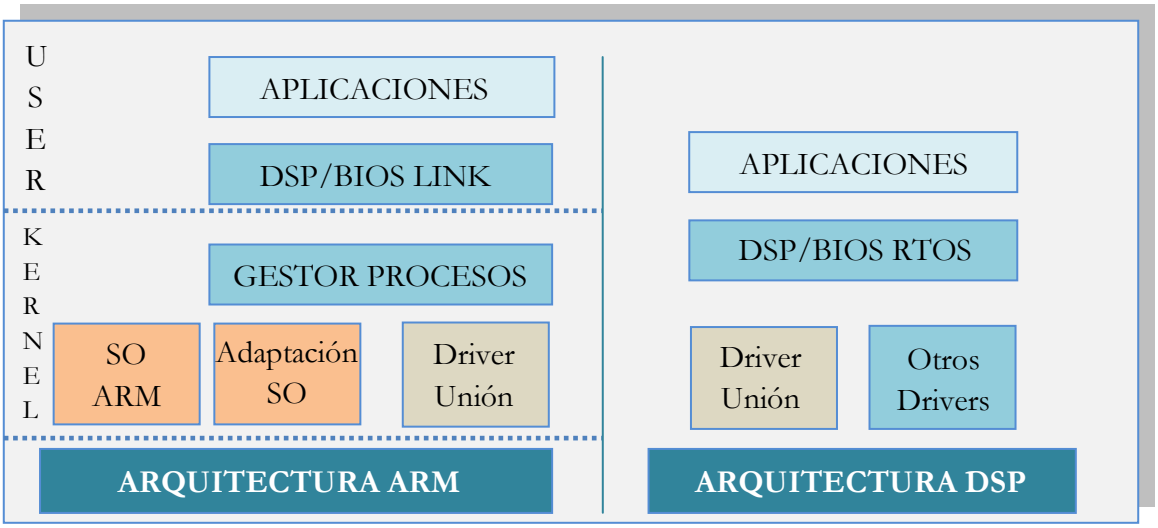

Fig. 13 Arquitectura de DSP/BIOS Link

En el lado izquierdo de la imagen se muestran los sub-módulos perteneciente a la arquitectura ARM, donde la herramienta ya de por sí asume que se estará ejecutando un sistema operativo (como sucede en este trabajo). Son los siguientes:

- Capa de adaptación del sistema operativo: permite a DSPLink abstraerse del sistema operativo particular sobre el que se ejecuta, facilitando enormemente su portabilidad.
- *Driver* de unión o enlazado: encapsula las operaciones de control de bajo nivel sobre el enlace físico que une ambos procesadores; además también se encarga de controlar la ejecución del DSP y la transferencia de datos entre procesadores.
- Gestor de procesos: actúa de interfaz entre sus capas anterior y posterior, esto es, proporciona las funciones de control del driver de unión al usuario a través de la

API de la siguiente capa. Esas funciones permiten controlar el DSP desde el ARM y entre ellas, cabe destacar las siguientes:

  - Carga del código del DSP.
  - Arranque del DSP para que éste comience a funcionar.
  - Localización de la cola de mensajes.

- API de unión o enlazado: representa la interfaz por la cual las aplicaciones acceden a los servicios que ofrecen los demás sub-módulos.

En el lado derecho de la imagen (ver Fig. 13) se muestran los sub-módulos de la arquitectura DSP, donde la estructura de capas se simplifica considerablemente respecto a la del ARM. Esto se debe, entre otros motivos, a dos características singulares: en primer lugar, a que las tareas del DSP (en el caso del sistema embebido empleado en este trabajo) sean concretas y "simples", es decir, el DSP recibe los datos y la orden de las operaciones a realizar y al terminar devuelve el resultado, quedando a la espera de recibir nuevas órdenes. De esta manera, el DSP actúa de esclavo respecto al procesador principal y todas las operaciones de gestión, administración y presentación de la información procesada son realizadas por el ARM. Como segunda característica singular está el sistema operativo del DSP, denominado DSP/BIOS [18], el cual es un sistema operativo de tiempo real (RTOS), de ahí que no exista una capa de adaptación del mismo.

Respecto a las diferentes fases por las que debe pasar una aplicación que emplee DSPLink, éstas son las siguientes:

- Fase de inicialización: conexión del ARM con el DSP y arranque de la ejecución de éste.
- Fase de ejecución: ejecución de las tareas particulares de cada sección, donde normalmente una o varias tareas de la sección ARM comunicarán datos con tareas de la sección DSP.
- Fase de finalización: desconexión ordenada de los procesadores y liberación de recursos empleados.

Dado que DSPLink está compuesto por diferentes componentes, es importante conocer y seleccionar los más apropiados en función de las necesidades de la aplicación a desarrollar. Para este trabajo en concreto, se han seleccionado aquellos que eran imprescindibles para cubrir las tareas de comunicación y sincronización entre núcleos ya que se busca reducir al máximo el volumen final de la misma eliminando aquellas parte innecesarias. Esto pone de manifiesto la elevada escalabilidad de DSPLink haciendo de ella una de sus grandes ventajas respecto a otras herramientas. En ocasiones, debido a la dependencia entre componentes, es posible que el empleo de uno de ellos en alguna de las fases implique el empleo de otros en fases diferentes, pero generalmente suelen ser independientes.

Tal y como se verá en el apartado 3.2.2.1, el proceso de configuración de DSPLink, mencionado anteriormente, es complejo e involucra a muchos elementos que conforman el sistema. Entre otras tareas, a través de ese proceso se configuran las diferentes secciones de la memoria de la tarjeta BeagleBoard y se generan una serie de librerías útiles para el proyecto final. Por tanto, una configuración incorrecta o equivocada implica un paso hacia atrás importante en el desarrollo de la aplicación final.

2.3.2.2 CMEM

CMEM es la herramienta *software* proporcionada nuevamente por el fabricante Texas Instruments empleada para solucionar los accesos a las diferentes secciones de memoria por parte de ambos núcleos del procesador OMAP3530. En primer lugar se van a comentar los motivos que obligan a emplear esta herramienta y posteriormente se describirán brevemente algunos detalles de la misma.

Cuando una aplicación que se ejecuta en el Linux embebido (núcleo 'ARM') reserva un determinado espacio de memoria, el sistema operativo es el encargado de reservar ese espacio, y lo reservará dentro del espacio que se indicó como disponible al inicio del sistema. Por tanto, esta reserva de memoria se hace de forma "desordenada", es decir, se realiza de acuerdo a criterios propios del sistema, y puede hacerlo en cualquier lugar del espacio de memoria de usuario (o del *kernel* si se especifica). Además, debido a las características de la reserva en memoria Linux, el espacio reservado no tiene por qué hacerse de forma físicamente continua, de manera que puede encontrarse partido en secciones y distribuido en distintas zonas.

En las aplicaciones en las que exista intercambio de información entre ARM y DSP (como sucede en este trabajo), será necesario compartir datos entre ambos procesadores, de manera que el DSP debe recorrer las posiciones de memoria donde el ARM sitúe los datos para que sean procesados. Aquí viene el primero de los inconvenientes, ya que el DSP necesita que los datos estén ubicados físicamente de forma continua en el espacio de memoria, algo que el ARM no lo garantiza. Además, para asegurar la integridad del sistema operativo, el DSP no debe acceder a posiciones de memoria que se encuentren dentro del espacio de memoria del sistema operativo GNU/Linux, ya sea espacio de memoria de usuario o del *kernel*, puesto que, si por algún motivo sucede un error en la aplicación y el DSP escribe en direcciones de memoria que se encuentran fuera de las estipuladas, se estarían destruyendo datos del propio sistema operativo y se corrompería todo el sistema. Hasta aquí los impedimentos del DSP a la hora de acceder a la información en memoria, pero el ARM también presenta uno, y es que cuando desde una aplicación que se ejecuta en este procesador, se intenta acceder a memoria que se encuentra fuera del espacio del sistema operativo, se producirá un error de acceso a memoria.

En definitiva, los inconvenientes descritos que plantean ambas arquitecturas hacen necesario el empleo de una herramienta adicional que solucione de algún modo la transferencia de información entre procesadores. Esa herramienta adicional es vista como un módulo más del *kernel* del sistema operativo Linux embebido en el núcleo ARM, y ha sido clave en este Trabajo Fin de Máster.

CMEM tiene la capacidad de reservar un espacio de memoria contiguo, fuera del espacio de memoria del sistema operativo y hacerlo accesible para el procesador donde se encuentra embebido dicho sistema. Esta disposición permitirá un flujo de información entre ambos procesadores seguro y eficaz. Además, el espacio de memoria reservado por esta herramienta ofrece la posibilidad de ser dividido en secciones (*pools*) de tamaño variable, de modo que el usuario pueda adaptar sus necesidades de memoria al tamaño del bloque. En el apartado 3.2.2.2 se describirá la configuración realizada en la aplicación desarrollada.

2.4 Entorno de desarrollo para aplicaciones 'ARM' y 'C64x+'

Code Composer Studio [19] (CCS en adelante) es el entorno de desarrollo integrado (IDE) del fabricante Texas Instruments empleado para editar y depurar aplicaciones construidas para los dos núcleos del procesador OMAP3530, siendo una de las mejores herramientas *software* para sistemas embebidos, de ahí su elección. Se trata de una versión muy actual (3 de Noviembre de 2011) ya que, aunque en el desarrollo inicial de este trabajo se comenzó con la versión 5.0.3, ésta era una versión de prueba hasta el lanzamiento de la definitiva versión 5.1.

2.4.1 Code Composer Studio v5.1

CCS está basado en la arquitectura *software* de código abierto Eclipse, la cual fue originalmente desarrollada como una arquitectura para la creación de herramientas de desarrollo pero que posteriormente se ha ido consolidando como la arquitectura estándar usada por los principales proveedores de sistemas embebidos. CCS combina las ventajas de la arquitectura abierta de Eclipse con las capacidades avanzadas de depuración de sistemas embebidos de Texas Instruments, dando como resultado un entorno para desarrolladores de sistemas embebidos. Aunque originalmente la arquitectura de CCS estuviera más enfocada al desarrollo de aplicaciones para DSPs, la versión más reciente de la herramienta de Texas Instruments permite la edicición y depuración de otras arquitecturas como ARM que presenta el procesador OMAP3530. La siguiente figura (ver Fig. 14) representa los principales elementos que conforman la arquitectura de CCS en aplicaciones para procesadores digitales de señal.

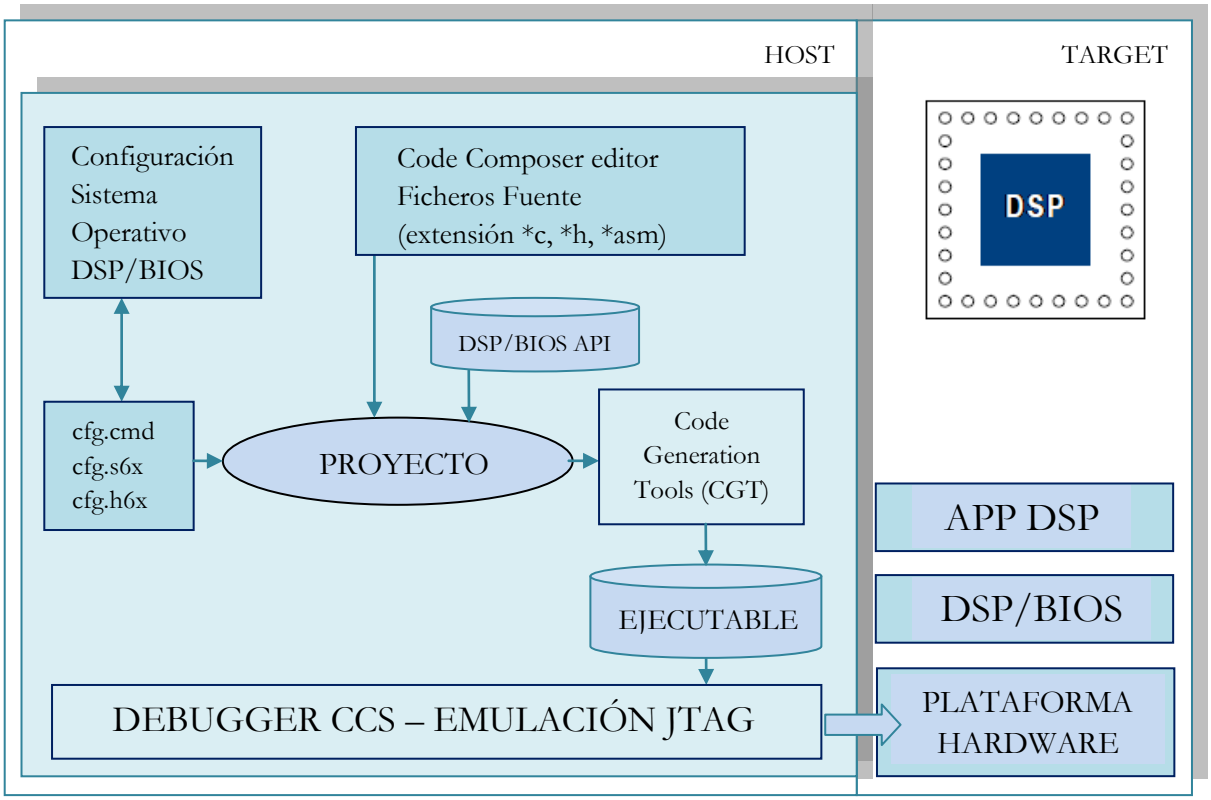

Fig. 14 Ciclo de desarrollo y testeo de aplicaciones para DSPs en Code Composer Studio

En la imagen, la aplicación para DSP requiere de un archivo de configuración **.tcf* [20] del sistema operativo DSP/BIOS donde se configuración los módulos a emplear del mismo, gestión de memoria, etc. y obviamente de los ficheros fuente necesarios que formen el proyecto. A continuación se emplean las *Code Generation Tools,* una herramienta de Texas Instruments que contiene las utilidades típicas para la construcción del ejecutable: compilador, ensamblador, optimizadores y lincador. Si el proceso se ha realizado con éxito, se puede iniciar la sesión de depuración típicamente a través de un emulador (para el caso de DSP) mediante la interfaz JTAG. A partir de este punto se pasa de trabajar en el *host* para pasar a la plataforma de desarrollo, sistema embebido, etc. donde se encuentre el procesador digital de señal. Finalmente, una vez establecida la comunicación, el desarrollador ya puede empezar a depurar su *software* ejecutándose en el procesador.

En cuanto a la apariencia de CCS (ver Fig. 15), el aspecto es el clásico que presenta un entorno de desarrollo de esta temática. En la parte superior presenta diferentes opciones para la edición del código, vista de elementos o configuración del proyecto. La barra gráfica, también en la parte superior cambia en función de la perspectiva en la que se trabaje: edición o depuración. En el caso de la primera, el siguiente icono  permite realizar las tareas de compilado, ensamblado y enlazado con un solo clic de ratón, mientras que el icono  permite lanzar el depurador, lo que hace que cambie la perspectiva a modo depuración. Son los iconos que más se emplean en la perspectiva de edición. Por otro lado, la perspectiva de depuración incluye los iconos típicos de reanudar , pausar y parar  la depuración además de los iconos que permiten avanzar instrucción a instrucción (*step to step*) entrando  o no  a depurar el código de funciones.

Por último, en el lateral izquierdo se encuentran los árboles de directorios de los proyectos construidos mientras que en la parte inferior se pueden anidar diferentes ventanas. En concreto, en este trabajo se han habilitado las pestañas de consola y problemas, ya que han sido las más útiles para las tareas desempeñadas, pero podrían ser éstas más algunas más, otras diferentes, etc. en función de las necesidades del usuario. La ventana de consola muestra información diferente en función de la perspectiva en la que se esté trabajando: en edición se puede visualizar el resultado de la ejecución de la cadena de compilación, ensamblado y lincado del proyecto mientras que en depuración, CCS permite ver los mensajes del código de la aplicación (impresión por pantalla *-fprintf-*) volcada en el DSP a través del emulador.

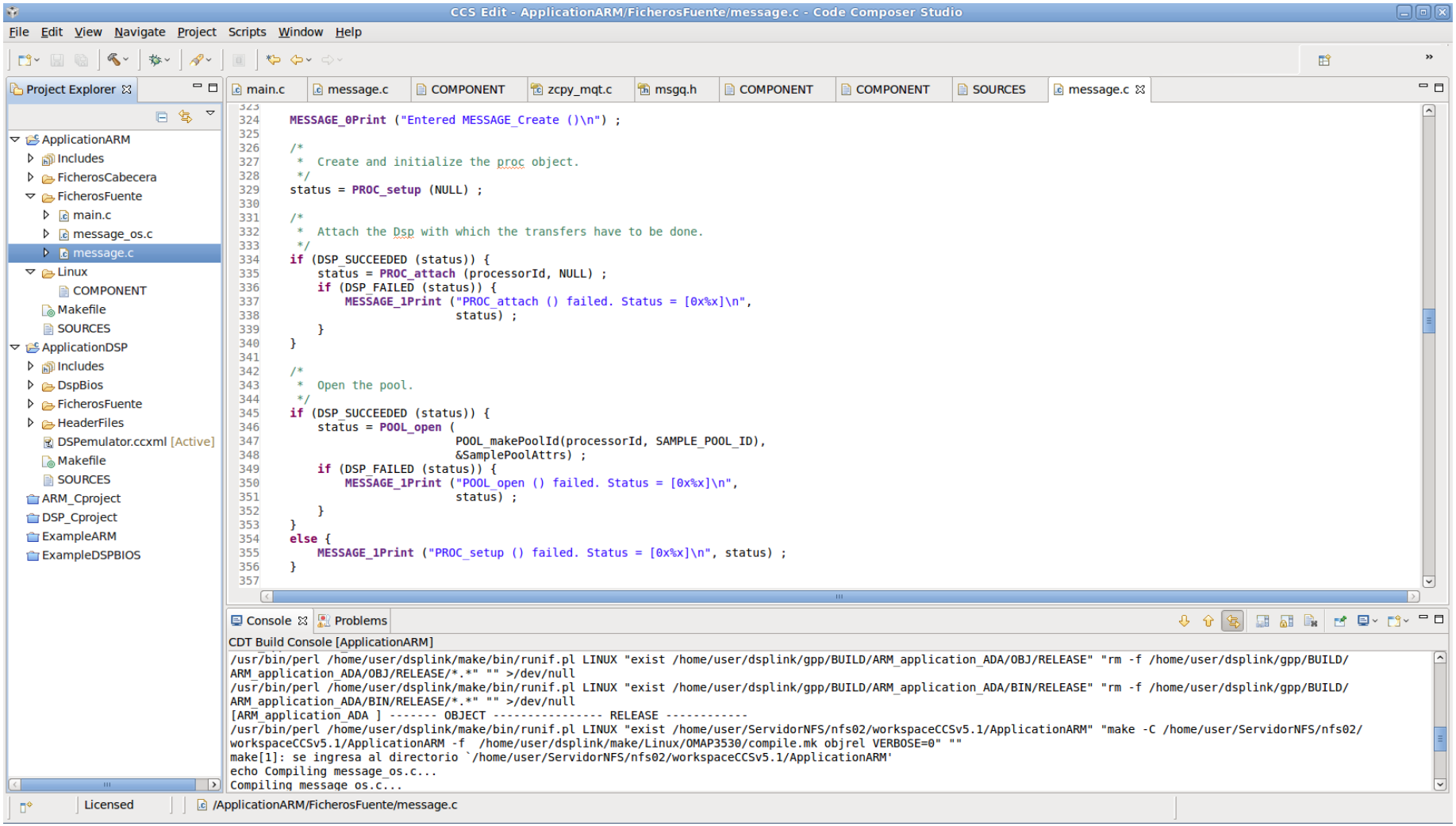

Fig. 15 Perspectiva del entorno de desarrollo integrado Code Composer Studio

Además, esta versión de CCS permite solucionar la falta de homogeneidad en el sistema operativo empleado para la edición y depuración de aplicaciones en ambos núcleos del procesador OMAP3530 que se venía realizando en el Grupo de Investigación GDEM. Anteriormente a la realización de este Trabajo Fin de Máster, el método de trabajo definido requería la acción de emplear Windows y Linux como sistemas operativos para realizar operaciones en el núcleo 'C64x+' y el núcleo 'ARM' respectivamente debido a la falta de soporte (controladores) de emuladores Blackhawk [21] en sistemas operativos Linux. Este hecho impedía trabajar bajo un único entorno para ambos núcleos con la problemática de tener que emplear dos ordenadores o la opción de una máquina virtual repartiendo recursos de computación entre el *host* y la propia máquina. En consecuencia, también se requerían dos herramientas *software* para la edición, compilación y depuración de las aplicaciones construidas, lo que implicaba tiempo para conocer dos entornos de desarrollo integrado diferentes (Code Composer Studio v3.3 - Windows y Eclipse - Linux) cada uno con sus particularidades y configuraciones. Dado que una de las principales utilidades de disponer de dos procesadores de estas tipologías en único *chip* como es el caso del OMAP3530 es poder comunicar y transferir información entre ellos para aligerar su procesamiento (entre otras tareas), la edición y depuración de aplicaciones para ambos núcleos adquiere una importancia muy significativa.

CCS en su versión 5.1 Linux ya ofrece soporte para los citados emuladores por lo que se consigue definir un nuevo método de trabajo más eficaz para trabajar sobre los dos núcleos del OMAP3530 bajo un único entorno de desarrollo y un único sistema operativo. En los apartados 3.3.2.1 y 3.3.2.2 se analizarán más en detalle los métodos de depuración empleados para 'ARM' y 'C64x+' respectivamente y alguna de las principales novedades que ha incorporado la nueva versión de CCS y que han supuesto una gran ventaja respecto a versiones anteriores.

2.5 Mplayer

MoviePlayer [22] (Mplayer en adelante) es el reproductor de vídeos empleado como base fundamental del trabajo desarrollado. Se trata de un reproductor multimedia multiplataforma bajo Licencia Pública General de GNU (GPL de GNU) con soporte para un amplio rango de codificadores/descodificadores (*codecs*) de audio y vídeo así como de controladores de salida para la representación del vídeo en pantalla. El motivo fundamental por el cual se ha seleccionado este reproductor para abordar el objetivo fundamental de este Trabajo Fin de Máster es el soporte de MPlayer para vídeos escalables, permitiendo descodificar vídeos con diferentes niveles de escalabilidad a través del descodificador OpenSVC que incorpora.

En la siguiente tabla (ver Tabla 1) se presenta información perteneciente a las diferentes opciones que soporta MPlayer en cuanto a los principales formatos de entrada, *codecs* de vídeo[5] y dispositivos de salida de vídeo.

Formatos de entrada	Codecs de vídeo	Disp. de salida (vídeo)
(S)VCD (Super Vídeo CD)	MPEG-1 y MPEG-2	X11 con extension SHM
CDRwin's .bin image file	MPEG-4 ASP (DivX, DivX4, DivX5, Xvid)	Xvmc (Xvideo Motion Compensation)
DVD/DVD encriptados	**MPEG-4 AVC (H.264, H.264/SVC)**	Vidix (VIDeo Interface for *niX)
Formato AVI	WMV 1/2	Cvidix (VIDIX on the console)
Formato ASF/WMV	WMV3	Dga (X11 DGA extensión)
Formato MP4/QT/MOV	RealVídeo 1.0, 2.0, 3.0	Gl (OpenGl renderer)
Formato RealVideo	Quicktime	**Fbdev (Framebuffer output)**
Archivos Ogg/OGM	DV video	Directfb (DirectFB support)
Matroska	Intel Indeo 4.1 y 5.0	Direct3d (Windows native Direct3D 9)
Formato VIVO	FLI/FLC	Aalib (text mode rendering)
Formato yuv4mpeg	VIVO 1.0, 2.0 y H.263	Svga (SVGAAlib output)
Streaming vía HTTP/FTP,RTP/RTSP	HuffYUV	Quartz (Mac OS X native)

Tabla 1 Fuentes de compatibilidad del reproductor multimedia MPlayer

[5] No se ha hecho referencia a los codificadores/descodificadores de audio por carecer de importancia en este trabajo, pero igualmente MPlayer ofrece una amplia gama de ellos.

La Tabla 1 muestra sombreada las opciones que han sido empleadas en este Trabajo Fin de Máster: ficheros con formato de entrada MP4 (*.*h264*), *códec* H.264/SVC para realizar la descodificación de vídeo mediante OpenSVC y controlador de salida Fbdev para visualizar los contenidos descodificados en un monitor a través de éste. La magnitud de la tabla, que no contiene la lista completa de opciones, pone de manifiesto el alto grado de flexibilidad y complejidad del reproductor, adaptándose a numerosos tipos de entornos de desarrollo.

Por otro lado, se han empleado dos versiones [23] de MPlayer diferentes en el desarrollo del trabajo: versión para PC y versión para el sistema embebido BeagleBoard. Esta dualidad de versiones se debe a que la versión PC proporciona entorno gráfico respecto a la versión de la tarjeta por lo que en ocasiones se ha recurrido a ella para interpretar mejor los comandos de Mplayer y otras tareas como la visualización de diferentes niveles de escalabilidad. El entorno gráfico está construido mediante la herramienta GTK (*GIMP Tool Kit*) y se denomina *GTKPlayer* [24].

La Fig. 16 muestra una vista del entorno gráfico de Mplayer en su versión para PC.

Fig. 16 Interfaz gráfico de Mplayer para entorno PC

En capítulos posteriores se analizará en profundida la configuración y estructura de MPlayer, un estudio que ha supuesto un trabajo significativo y de especial importancia para este Trabajo Fin de Máster.

2.6 Estándar de codificación de vídeo escalable (SVC)

Los inicios del estándar SVC (*Scalable Vídeo Coding*) [25] comienzan en 2003, con una solicitud de modelos de codificación de vídeo escalable a la comunidad científica '*Call for Proposals on SVC Technology*'. La necesidad de desarrollar un nuevo estándar de codificación de vídeo surge debido al crecimiento exponencial que han experimentados los terminales móviles con capacidad de reproducir contenidos multimedia en los últimos años. Estos terminales presentan diferentes características y limitaciones (tamaño de pantalla, capacidad de procesamiento, consumo de energía) que hacen que el sistema de vídeo tradicional sea ineficiente. Además, para la transmisión de vídeos sobre redes de transmisión tan heterogéneas como Internet (*streaming* de vídeo), el sistema de codificación de vídeo necesita proveer el *bitstream* al cual debe ser descodificado para reconstruir la señal de vídeo a varios niveles de calidad, resolución espacial y resolución temporal, todos ellos con calidad suficiente, algo posible solo mediante un sistema de codificación escalable.

Por tanto, el objetivo que persigue el estándar de codificación escalable es el de transmitir un único flujo de vídeo que lleve embebidos diferentes niveles de información correspondientes a una única secuencia de vídeo, de modo que, una vez descodificado dicho flujo, el descodificador sea capaz de proporcionar la misma secuencias de vídeo con diferentes calidades. Estas calidades dependerán de la capacidad de cada receptor, de las opciones preferidas por los usuarios y de la capacidad variable del canal de transmisión.

Para ello, una primera solución pasaría por comprimir y almacenar cada una de las secuencias de vídeo codificadas a diferentes calidades, pero esta técnica introduciría una gran sobrecarga de almacenamiento y es inviable para aplicaciones en tiempo real, algo muy demandado en la actualidad. La solución alternativa y definitiva es la que permite al usuario seleccionar dinámicamente los siguientes parámetros: *bit-rate,* resolución espacial y resolución temporal (número de imágenes por segundo), de forma que se elimina la citada sobrecarga y se incrementa la flexibilidad y funcionalidad del estándar. Esta segunda solución presentada se denomina escalabilidad de vídeo y en el estándar H.264/SVC se definen tres tipos [26]:

- Escalabilidad espacial: aquella señal de vídeo escalable espacialmente que puede ser descodificada y posteriormente reproducida con diferentes resoluciones espaciales.

- Escalabilidad temporal: aquella señal de vídeo que puede ser descodificada y posteriormente reproducida con diferentes *frame rates* (frecuencias de imagen, en número de imágenes por segundo).

- Escalabilidad SNR (*Signal Noise Ratio*) o de calidad: aquella señal de vídeo que está codificada con escalabilidad SNR (también denominada escalabilidad de calidad) y que puede ser descodificada y posteriormente reproducida con distintos grados de fidelidad o de calidad.

Las diferentes escalabilidades son codificadas en lo que el estándar H.264/SVC denomina capas, de forma que se comienza con la capa base, la cual contiene el menor nivel de resolución espacial, temporal y de calidad y se continua por aquellas que incrementan los niveles de escalabilidad, denominándose capas de mejora. Sin embargo, hay que resaltar que

el proceso de codificación de cada una de las capas (salvo la capa base) se apoya en la información resultante de la codificación de la capa inmediatamente inferior, de manera que se omite cierta redundancia de información y se incrementa la eficiencia en cuanto a ancho de banda empleado, otra ventaja del estándar frente al resto de codificadores tradicionales.

Un ejemplo gráfico de los tres tipos de escalabilidades se presenta en la siguiente figura (Fig. 17):

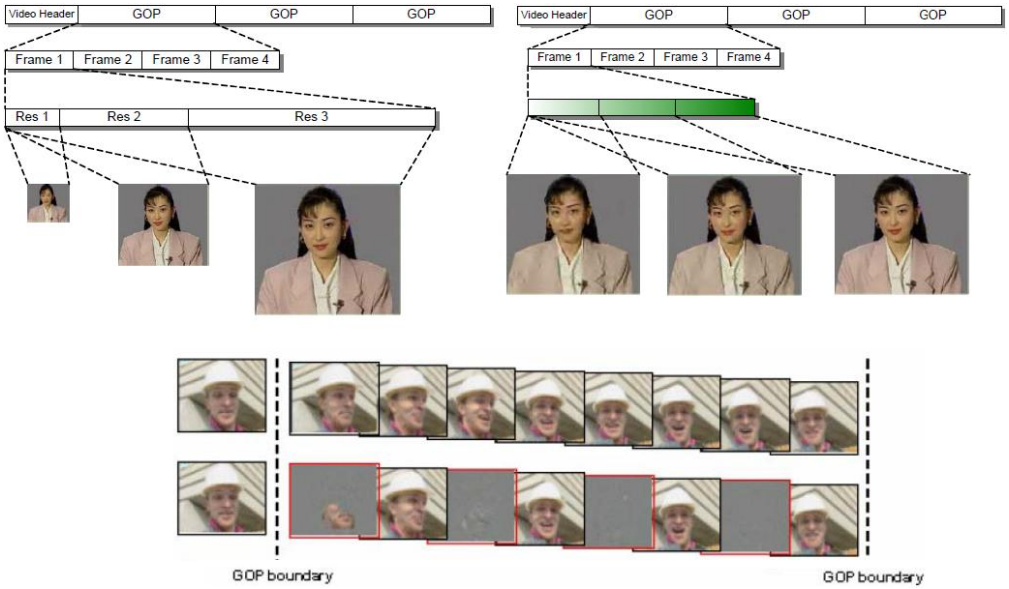

Fig. 17 Tipos de escalabilidad del estándar de vídeo escalable H.264/SVC

Una vez vistas las necesidades de un nuevo estándar así como su elemento diferenciador, la escalabilidad, se va a presentar un esquema simplificado de un codificador H.264/SVC, compuesto por tres módulos resultantes de variaciones de un codificador H.264 y que se encuentran interconectados entre sí de tres formas diferentes (ver Fig. 18). La primera conexión entre los módulos hace que la secuencia de entrada de todos ellos sea la misma, pero con resoluciones espaciales diferentes obtenidas tras la acción de los módulos de diezmado (bloque denominado "Reducción Espacial" en la Fig. 18). La segunda conexión se realiza internamente, mediante un mecanismo conocido como predicción *inter-layer*. Esta técnica sirve para optimizar el flujo de datos, ya que comunica a las capas superiores la información resultante de la codificación realizada en la capa inferior, de manera que las capas superiores aprovechen las predicciones ya realizadas por las capas inferiores en el proceso de su propia predicción para acelerar la codificación, explotando al máximo las redundancias espaciales y temporales. En este proceso también se introduce un pequeño error denomina deriva (*drift)* que mide la diferencia existente entre la salida de vídeo escalable que proporciona un descodificador H.264/SVC y las salidas que proporcionarían varios descodificadores H.264 con la misma entrada, puestos en paralelo y configurados con los mismos parámetros. Por último, la tercera conexión se realiza en la salida, donde se produce la multiplexación de los diferentes codificadores H.264, de forma que el *bitstream* de salida contendrá embebidas la información de la capa base así como de las capas de mejora.

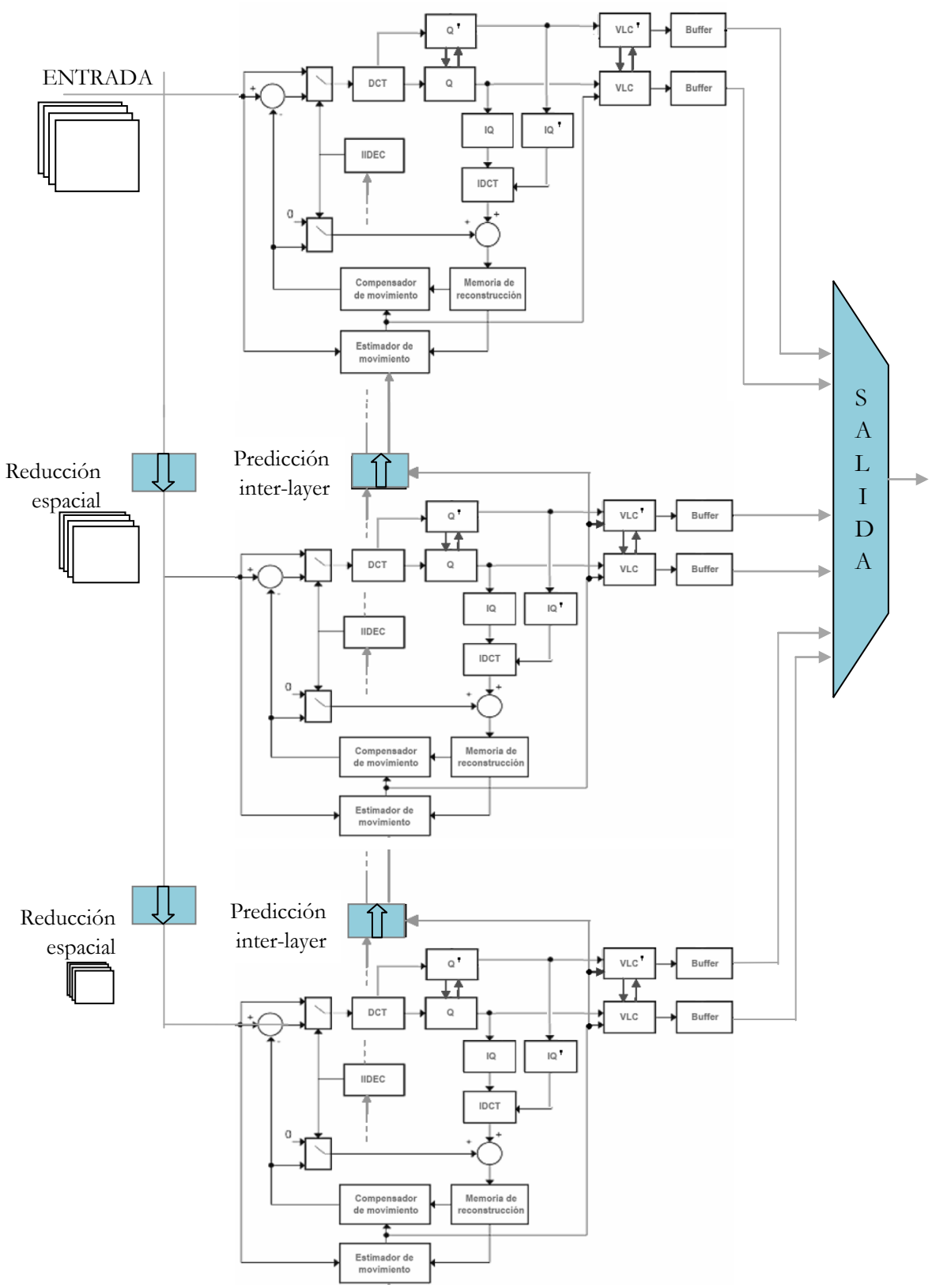

Fig. 18 Arquitectura simplificada de un codificador H.264/SVC

2.6.1 Descodificador OpenSVC

Una vez vista la arquitectura simplificada de un codificador escalable y analizadas sus principales diferencias con su predecesor, se va a presentar uno de los principales elementos de este Trabajo Fin de Máster, el descodificador de vídeo escalable OpenSVC [27], conforme al estándar H.264/SVC e implementado en lenguaje C.

El descodificador implementa el perfil *Scalable Baseline* e incluye herramientas que permiten realizar cambios en cuanto a escalabilidades espacial, temporal y de calidad. Está basado en el perfil *Baseline* del estándar H.264/AVC y además incluye la mayoría de herramientas del perfil *Main* a excepción de la codificación entrelazada y la predicción ponderada debido a la complejidad que supone implementar estas técnicas para sistemas embebidos. Pese a ello, el *software* de referencia OpenSVC es adaptable a este tipo de sistemas, de modo que su análisis e integración en la aplicación final han sido favorecidos por esta característica.

OpenSVC proporciona soporte para la descodificación parcial del bitstream, permitiendo elegir las capas que se desean descodificar de forma independiente y específica. Esto supone una gran ventaja frente al *software* de referencia del proyecto de codificación de vídeo escalable JSVM (*Joint Scalable Video Model*) [28], debido a que éste únicamente permite descodificar la capa superior del bitstream, es decir, la capa de mejora con la mayor escalabilidad espacial, temporal y de calidad. Esta característica hace que el descodificador OpenSVC presente mayor flexibilidad y capacidad de adaptación sobre diferentes plataformas de desarrollo mediante la selección de la capa adecuada con el fin de obtener una descodificación en tiempo real.

La selección y cambio de capa a descodificar también se puede realizar durante el propio proceso de descodificación, ya que OpenSVC contiene varios mecanismos para llevarlo a cabo mediante comandos específicos. Además, si se produce algún error en la transmisión y afecta a capas de mejora, el descodificar es capaz de cambiar de capa de forma autónoma y continuar el proceso de descodificación.

En la siguiente figura (ver Fig. 19) se muestra el flujo de datos que se produce en el proceso de descodificación cuando la capa superior de una secuencia de cuatro capas no se descodifica.

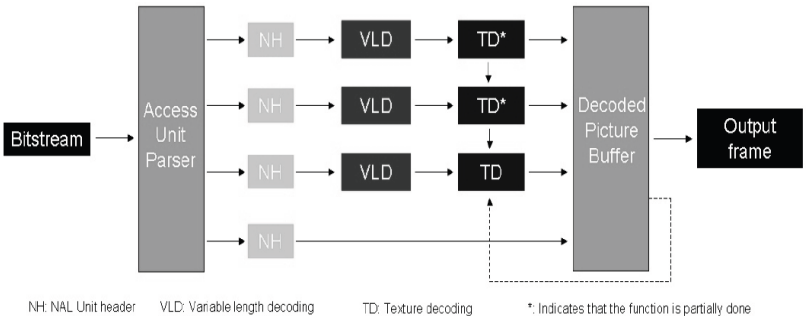

Fig. 19 Flujo de datos en el proceso de descodificación de un vídeo escalable

Tal y como refleja la imagen, los procesos de descodificación de longitud variable y descodificación de textura solo se llevan a cabo en las tres capas inferiores descartando la capa superior, realizándose en consecuencia una descodificación parcial del *bitstream*.

A continuación se presenta un diagrama de flujo simplificado del descodificador OpenSVC (ver Fig. 20), en el cual se muestra el comportamiento del descodificador ante la recepción de un *bitstream*.

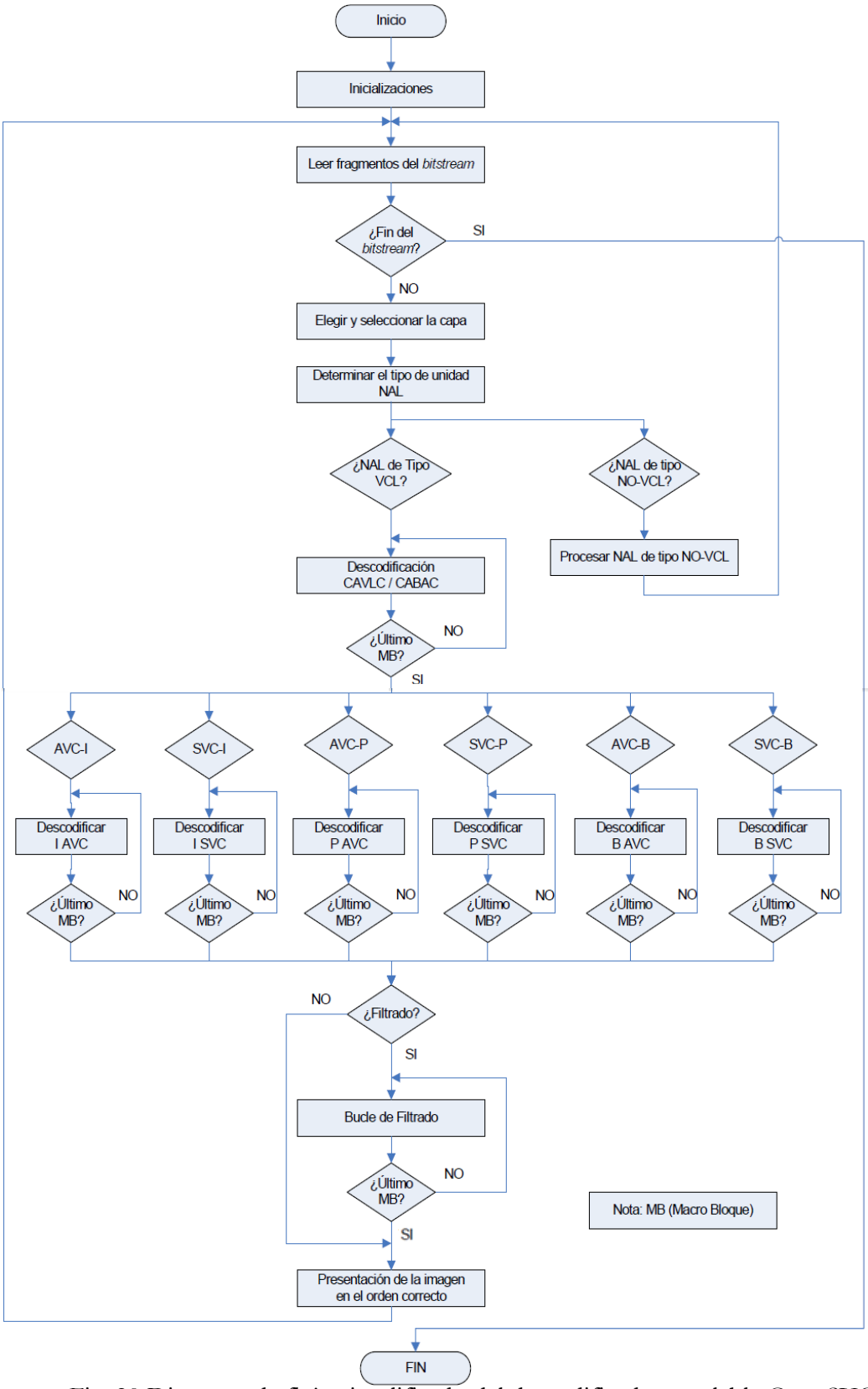

Fig. 20 Diagrama de flujo simplificado del descodificador escalable OpenSVC

En la Fig. 20 se muestra como el descodificador lee una secuencia de vídeo escalable perteneciente a un *bitstream* H.264/SVC e identifica las distintas unidades NAL[6] de la secuencia. Después de descodificar la cabecera de la NAL, se identifica si es de tipo VCL[7] o no VCL. A continuación, si la unidad NAL contiene una porción de interés de la capa seleccionada, el descodificador extrae todos los elementos sintácticos del *bitstream* y los almacena en *buffers* intermedios. Si la NAL procesada se debe presentar en pantalla, cada macrobloque estará completamente descodificado, pero si no se puede presentar en pantalla, entonces el macrobloque estará solo parcialmente descodificado. Las imágenes descodificadas se almacenarán en el orden correcto de presentación y se presentarán mediante la librería SDL (PC *Simple Direct Media Layer Library*).

Por último, mencionar que existen principalmente dos versiones de OpenSVC: una para entorno PC y otra para procesadores digitales de señal del fabricante Texas Instruments, particularidad que hace que la plataforma de desarrollo empleada en este trabajo sea idónea. Además, la versión para DSPs del descodificador ha sido desarrollada y optimizada en el Grupo de Investigación GDEM [29], por lo que se cuenta con un notable grado de conocimiento y experiencia en este área. Por otro lado, la primera de las versiones de OpenSVC, dirigida a PCs presenta como principal y significativa diferencia respecto a la versión DSP la interfaz gráfica que proporciona al usuario. No obstante, en el apartado 4.3.4 se llevará a cabo un análisis más profundo del descodificador OpenSVC en su versión DSP describiendo en detalle su estructura y funcionamiento.

[6] La organización de los datos codificados se estructura dentro de unidades NAL (Network Abstraction Layer), que son paquetes formados por un número definido de bits y que tienen por objetivo adaptar la información codificada a las características del medio de transmisión. Cada NAL presenta un número de identificación, el tipo de NAL y los datos.

[7] Las unidades VCL (*Video Coding Layer*) transportan la información correspondiente al vídeo codificado mientras que las unidades no VLC transportan información adicional necesaria en el proceso de descodificación.

3 CONFIGURACIÓN SISTEMA

En este capítulo se presentan las configuraciones realizadas en los distintos elementos de la aplicación desarrollado a lo largo de este Trabajo Fin de Máster.

En primer lugar se realiza una breve descripción de las cadenas de herramientas necesarias para obtener los archivos ejecutables correspondientes a los dos núcleos del procesador OMAP3530, más conocidas como toolchains.

Posteriormente, se lleva a cabo un análisis en profundidad de la configuración de los módulos DSPLink y CMEM, comenzando por sus procesos iniciales y terminando en la obtención e inmediata carga en el kernel Linux embebido del núcleo 'ARM'.

A continuación, se describen los métodos empleados para realizar la edición y depuración de aplicaciones tanto para el núcleo 'ARM' como para el 'C64x+', haciendo especial hincapié en el nuevo método de depuración multiprocesador que incorpora la nueva versión de Code Composer Studio.

Por último, se presentan una serie de scripts de configuración que permiten agilizar el desarrollo del trabajo automatizando operaciones repititivas e incrementando la eficiencia.

3.1 Entorno de Compilación - *Toolchains*

Antes de analizar la configuración de los distintos módulos empleados, se van a describir los entornos de compilación empleados que permiten obtener los ficheros ejecutables para los núcleos del procesador OMAP3530 a través de un conjunto de herramientas (*toolchains*) compuesto por compilador, ensamblador y enlazador entre otros. Este orden en la presentación de contenidos se debe a que en el proceso de configuración del módulo DSPLink ya es necesario recurrir a ellas, por lo que parece inevitable no mencionarlas como punto de partida en la configuración del sistema.

En primer lugar, mencionar que el método de trabajo por el cual se obtienen ambos ejecutables difiere notablemente debido a la poca similitud entre arquitecturas ARM y DSP. Por tanto, para la arquitectura ARM será necesario disponer de un conjunto de herramientas que permitan obtener un ejecutable para ARM desde una arquitectura con procesador x86 Intel, esto es, un entorno de compilación cruzada a través de la cadena de herramientas empleada. Para este Trabajo Fin de Máster se ha elegido la *toolchain* del fabricante Codesourcery [30] para arquitecturas ARM, en concreto su versión más actual (2010).

Un esquema del entorno de compilación cruzada que se propone se muestra a continuación (ver Fig. 21):

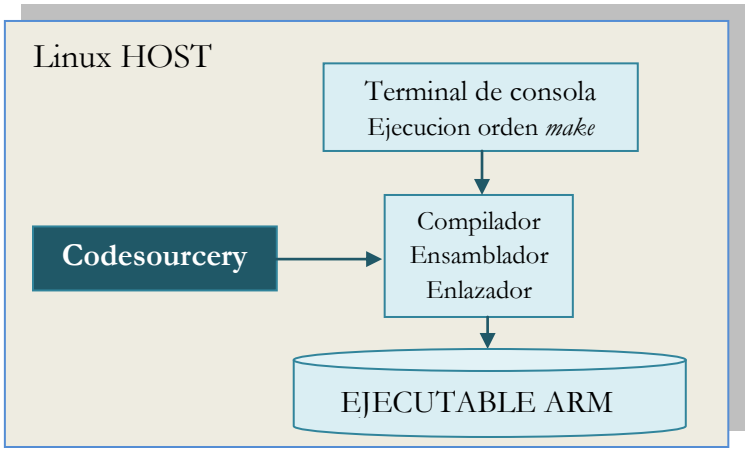

Fig. 21 Entorno de compilación cruzada para el núcleo 'ARM'

La versión elegida de Codesourcery es muy completa ya que ofrece numerosas utilidades además de las comentadas para compilación: *make, binutils, autotools gdb,* etc. Más adelante, cuando se definan los métodos de depuración empleados, se volverá a hacer referencia a esta herramienta.

A través de una configuración sencilla en el PC de desarrollo Linux [31], la cadena de herramientas de Codesourcery queda instalada y lista para usarse, de tal forma que cada vez que se vaya a lanzar una sesión de compilación, se deben ejecutar el siguiente comando:

```
$ make ARCH=arm CROSS_COMPILE=arm-none-linux-gnueabi-2010.09-
```

La primera variable, ARCH, indica al compilador que código debe utilizar en caso de que exista código para varias arquitecturas, mientras que la segunda variable,

CROSS_COMPILE informa a la utilidad make qué compilador se va a emplear. También es posible que ambas variables queden configuradas de forma interna, de manera que el usuario únicamente tenga que escribir *make* en el directorio específico para lanzar el proceso de generación del ejecutable, tal y como se ha realizado en este Trabajo Fin de Máster.

En cuanto a la cadena de herramientas para el núcleo 'C64x+', se han empleado las denominadas 'Code Generation Tools', en su versión C6000, del fabricante Texas Instruments tal y como se comentó en el apartado 2.4.1 del capítulo anterior. Al tratarse del mismo fabricante, el entorno de desarrollo integrado Code Composer Studio también proporciona estas herramientas en su estructura de directorios. Sin embargo, se decidió emplear esta utilidad descargándola a través de la página web oficial del fabricante debido al desconocimiento de la primera de las opciones planteadas. Posteriormente, en el proceso de configuración del módulo DSPLink, se corroboró el funcionamiento de la *toochain*.

Respecto al conjunto de fases necesarias para obtener el fichero ejecutable, ya sea para ARM o DSP, las diferencias entre ambos ahora sí que son escasas, en contraposición a lo que se comentaba al inicio de este apartado acerca del entorno de compilación cruzada. En consecuencia, el ciclo de creación de ejecutable está compuesto principalmente por las siguientes fases:

- Compilador C/C++: encargado de transformar el código fuente en lenguaje C o C++ de la aplicación a lenguaje ensamblador, a la vez que puede realizar optimización de ese código en diferentes niveles.

- Ensamblador: encargado de generar los archivos objeto en código máquina a partir del código fuente en ensamblador.

- Enlazador: encargado de generar un único módulo objeto ejecutable a partir de los archivos objeto generados anteriormente y otros pertenecientes a librerías si los hubiera.

A través del entorno de desarrollo CCS, todo este ciclo descrito se puede simplificar en un solo paso mediante la utilidad *build*, acelerando el proceso y liberando al usuario de realizarlo en diferentes pasos.

3.2 Gestión de memoria y generación de módulos

En este apartado se van a presentar dos aspectos relacionados con la configuración del sistema diferentes pero muy ligados entre sí: la gestión de la memoria del sistema embebido BeagleBoard y la configuración y generación de los módulos DSPLink y CMEM. Es de significativa importancia realizar ambas configuraciones correctamente para que las aplicaciones desarrolladas para los núcleos del procesador OMAP3530 funcionen como deberían.

Al ser el tema de la configuración de la memoria de la BeagleBoard un aspecto complejo y problemático debido a que involucra a muchas partes, se va a realizar en primer lugar una descripción del mapa de memoria del sistema embebido empleado en este Trabajo Fin de Máster, detallando los rangos de cada sección en direcciones de memoria y sus respectivos tamaños. A continuación, se analizarán en detalle los procesos de configuración de los módulos DSPLink y CMEM y por último se describirán las principales técnicas existentes para insertarlos en el *kernel* Linux embebido en el núcleo 'ARM' del procesador OMAP3530.

3.2.1 Mapa de memoria del sistema embebido BeagleBoard

Para la realización de este Trabajo Fin de Máster se ha empleado la versión C4 del sistema embebido BeagleBoard, el cual dispone de 256 MB de memoria. A priori, esta memoria debe tener al menos capacidad suficiente para instalar un sistema operativo Linux embebido (distribución Angström), alojar el programa construido para el DSP y permitir las comunicaciones entre los dos núcleos del procesador OMAP3530. Dicha comunicación hace referencia a los módulos empleados, los que requieren determinadas secciones de memoria para realizar su funcionalidad.

Por tanto, la primera sección del mapa de memoria de la BeagleBoard será para el sistema operativo Linux ejecutándose en el núcleo 'ARM'. Su tamaño debe ser suficientemente grande para la ubicación en esa sección de *buffers* de vídeos requeridos por el visor MPlayer en el inicio de su ejecución. De lo contrario, la configuración de memoria no generará ningún error inicialmente, pero la ejecución de la aplicación, una vez construida, se abortará sin posibilidad de continuar. Además, es importante conocer que el *kernel* Linux empleado emplea alrededor de 13 MB respecto de la cantidad total reservada para procesos y configuraciones internas, por lo que hay que seleccionar este parámetro adecuadamente. En este trabajo se han establecido diferentes tamaños[8] para esta sección debido a diferentes pruebas realizadas, pero finalmente se fijaron 105 MB para el sistema operativo garantizando el espacio requerido por las tareas iniciales de MPlayer comentadas anteriormente.

A continuación, se ha dejado sin emplear una cantidad de memoria alrededor de 7 MB que se puede usar para la inserción de nuevos módulos (aparte de los empleados en este

[8] Se puede modificar la cantidad de memoria reservada para el sistema operativo Linux embebido antes del arranque de éste, en el entorno *uboot*, configurando la variable de entorno `setenv otherargs_oha 'mem=xxxxM@0x80000000'` donde xxxx indica la cantidad de memoria expresada de forma numérica.

trabajo, DSPLink y CMEM) o futuras ampliaciones de la aplicación desarrollada. Precisamente, esa sección de 7 MB ha permitido trabajar con diferentes rangos para el módulo CMEM, ya que éste necesita de un tamaño suficiente para albergar los datos que se transfieren entre los procesadores del OMAP3530. Se ha optado por configurar un tamaño de 5 MB para el módulo CMEM, más que suficiente para el volumen de datos a transferir, como se verá en el apartado 3.2.2.2, pero que aporta al sistema en conjunto mayor seguridad a la hora de descartar fallos de memoria.

Inmediatamente después de la sección de CMEM, se encuentran de forma consecutiva las secciones pertenecientes al módulo DSPLink. De hecho, todas las secciones definidas restantes pertenecen a diferentes elementos de DSPLink y algunas de ellas tienen un tamaño impuesto por el fabricante de la herramienta, pero sí es posible ubicarlas libremente en función de las necesidades del usuario siempre y cuando no se produzccan solapamientos entre ellas. Existen dos bloques denominados Vectores de Reset del DSP y DDR2[9] que son de imprescindible configuración para el empleo de DSPLink. El primero de ellos tiene un tamaño fijo de 128 bytes y almacena la dirección de memoria donde se encuentra el código ejecutable de la aplicación para el núcleo 'C64x+'. También acudirá a esta sección cuando interactúe por primera vez en una aplicación que comunica los núcleos 'ARM' y 'C64x+'. Por otro lado, la sección DDR2 tiene un tamaño variable y se emplea para guardar el código ejecutable del 'C64x+' así como las variables y estructuras de datos utilizadas. Ese tamaño debe ser superior al tamaño del propio ejecutable del 'C64x+' por lo que se ha optado nuevamente por emplear un tamaño que no de lugar a problemas de memoria, 128 MB en esta ocasión. En este caso particular, la elección de este tamaño está condicionada por la configuración que se realizó en proyectos anteriores dentro del Grupo de Investigación GDEM, por lo que se prefirió mantener esa estructura[10].

Por último, existen tres bloques más que se corresponden con las secciones DSPLinkMEM y DSPLinkMEM1 (192 KB en total) agrupadas en un único bloque y empleadas para los ejemplos que proporciona DSPLink en su estructura de ficheros y con la sección POOL_MEM (832 KB) necesaria para los métodos de sincronización y transferencia entre procesadores. Es en esta última sección donde se almacenarán los mensajes de reducido tamaño intercambiados entre los dos núcleos del procesador OMAP3530 a través de colas de mensajes y que serán analizados en el apartado 4.3.1.

La Fig. 22 muestra el mapa de memoria del sistema embebido BeagleBoard con sus secciones definidas en cuanto a tamaño y rangos de direcciones de memoria empleados proporcionando una visión completa del análisis realizado.

[9] No guarda relación con el tipo de memoria RAM.

[10] El tamaño de la memoria DDR2 así como su ubicación dentro del mapa de memoria de la BeagleBoard puede ser modificado fácilmente ya que se conocen las técnicas empleadas para ello. De hecho, se han realizado pruebas con diferentes tamaños de DDR2 antes de fijar el tamaño final.

Rango de direcciones	Bloque (tamaño)
0x8000 0000	LINUX (105 MB)
0x8580 0000	SIN EMPLEAR (38 MB)
0x8790 0000	CMEM (5 MB)
0x87E0 0000	DSPLink VECTORES DE RESET (128 B)
0x87E0 0080	DDR2 (1 MB)
0x87F0 0000	DSPLink MEM /DSPLink MEM1 (192 KB)
0x87F3 0000 0x8800 0000	POOL_MEM (832 KB)

Fig. 22 Secciones de memoria del sistema embebido BeagleBoard

3.2.2 Configuración y generación de módulos

Una vez establecidas las diferentes secciones que debe contener el mapa de memoria del sistema embebido, hay que escribir esa configuración en los ficheros correspondientes dentro de DSPLink y CMEM para que tenga efecto. A continuación, a través de los siguientes dos apartados, se analizarán en detalle esas configuraciones.

3.2.2.1 DSP/BIOS Link

La correcta configuración y obtención [32] del módulo DSPLink han supuesto un esfuerzo notable dada su extensa y compleja estructura de directorios y su particular modo de generar los ejecutables. Por ello, se van a describir en detalle las diferentes fases necesarias para la generación del módulo *dsplinkk.ko* y sus correspondientes librerías, haciendo referencia a las *toolchains* mencionadas en el apartado 3.1.

3.2.2.1.1 Configuración del entorno *make*

El módulo DSPLink, al igual que la mayoría de los módulos manejados a lo largo de este Trabajo Fin de Máster requiere de la utilidad *make* para su generación. El objetivo de esta utilidad es determinar automáticamente qué módulos de un programa necesitan ser recompilados y ejecutar las órdenes apropiadas para esa tarea. Antes de usar esta orden, se requiere de un fichero denominado *makefile* que describe las relaciones entre los ficheros que componen el programa, así como las órdenes necesarias para actualizar cada uno de ellos.

En la estructura de ficheros de DSPLink existe un directorio denominado *make* que establece clasificaciones entre arquitecturas (ARM o DSP), más adelante entre sistemas operativos empleados (Linux o DSP/BIOS) y posteriormente entre dispositivos (OMAP3530, DaVinci, DM357, etc.). Sin entrar en los detalles de la estructura de directorios de la utilidad *make*, ya que no es objetivo de este trabajo, es necesario configurar

una serie de ficheros (ficheros de distribución salvo *Rules.mk*) tanto para ARM como para DSP donde se les indique la cadena de herramientas, *toolchain*, empleada para cada procesador además de otros parámetros.

Es importante anotar que esta configuración se realiza para obtener los ejecutables para arquitecturas ARM y DSP a través de la estructura de DSPLink, pero en este Trabajo Fin de Máster se han empleado otras técnicas para ello, que serán analizadas en el apartado 3.3. Sin embargo, todas las configuraciones mencionadas a continuación son necesarias para generar el módulo asociado a DSPLink.

En la siguiente tabla (ver Tabla 2) se muestra el nombre de esos ficheros, su ubicación en la estructura de ficheros de DSPLink, el procesador al que va dirigido y algunos de los campos más representativos modificados como por ejemplo las rutas donde se encuentran las *toolchains* tanto de ARM y DSP así como la ruta donde se encuentran los ficheros fuente del *kernel*, necesario para obtener los módulos correspondientes para esa versión.

NÚCLEO	Fichero(ubicación DSPLink)	Campos destacados
ARM	omap3530_2.6.mk $(DSPLINK)/make/ Linux/	BASE_BUILDOS:=${HOME}/Beagle/Validation/sources_git BASE_TOOLCHAIN:= /opt/codesourcery/arm-none-linux-gnueabi/arm-2010.09 BASE_CGTOOLS:= $(BASE_TOOLCHAIN)/bin COMPILER := $(BASE_CGTOOLS)/arm-none-linux-gnueabi-2010.09-gcc LINKER := $(BASE_CGTOOLS)/arm-none-linux-gnueabi-2010.09-gcc CROSS_COMPILE := arm-none-linux-gnueabi-2010.09-
ARM	Rules.mk[11] $(DSPLINK)/gpp/ src/	KERNEL_DIR:=home/user/Beagle/Validation/sources_git TOOL_PATH := /opt/codesourcery/arm-none-linux-gnueabi/arm-2010.09/binMAKE_OPTS = ARCH=arm CROSS_COMPILE=$(TOOL_PATH)/arm-none-linux-gnueabi-2010.09-
DSP	c64xxp_5.xx_linux.mk $(DSPLINK)/make/ DspBios/	BASE_INSTALL := /opt/TI BASE_SABIOS := $(BASE_INSTALL)/bios BASE_BUILDOS :=$(BASE_SABIOS)/packages/ti/bios XDCTOOLS_DIR := /opt/TI/xdctools_3_22_02_27 BASE_CGTOOLS := $(BASE_INSTALL)/c6000 BASE_CGTOOLSBIN:= $(BASE_CGTOOLS)/bin COMPILER := $(BASE_CGTOOLSBIN)/cl6x LINKER := $(BASE_CGTOOLSBIN)/cl6x -z TCONF := $(BASE_TCONF)/tconf

Tabla 2 Resumen de la configuración del entorno *make*

3.2.2.1.2 Configuración del entorno de DSPLink

Esta configuración es simple pero de transcendental importancia ya que se configuran un par de variables de entorno requeridas posteriormente por DSPLink. Las variables se muestran a continuación (ver Tabla 3):

[11] En este fichero existen diferentes plataformas sobre las cuales se realiza la configuración del entorno make. Se deben modificar únicamente los campos reflejados en la tabla para la plataforma OMAP3530.

Variable de entorno	Contenido
DSPLINK	`$(HOME)/dsplink`
PATH	`$(DSPLINK)/etc/host/scripts/Linux`

Tabla 3 Configuración del entorno de DSPLink

La variable de entorno DSPLINK define el directorio raíz para la instalación de DSPLink mientras que la variable de entorno PATH apunta al directorio donde se encuentran los *scripts* proporcionados en la instalación. Se recuerda que cualquier modificación en cuanto a movimiento de carpetas a otros directorios obliga a actualizar estas variables de entorno.

La configuración de estas dos variables se realiza a través del fichero `dsplinkenv.bash`, ya existente dentro de la estructura de directorios de DSPLink. Su ubicación es la siguiente: `/home/user/dsplink/etc/host/scripts/Linux/dsplinkenv.bash`. Esta operación se realiza desde una terminal de consola del *host* Linux en modo usuario. Es importante distinguir en Linux el rol que se tiene para realizar las operaciones ya que hacerlas en modo superusuario (*root*) o usuario normal puede acarrear problemas en los directorios configurados anteriormente. Una vez realizados los cambios en el fichero `dsplinkenv.bash`, se obtiene en el terminal el resultado de la configuración.

3.2.2.1.3 Configuración de DSPLink

El proceso de configuración de DSPLink se realiza a través de un *script* escrito en *perl*[12], que tras ejecutarlo, esperará en línea de comandos las diferentes opciones para configurar el módulo. Mediante la siguiente tabla (ver Tabla 4) se muestra la configuración realizada en el trabajo desarrollado.

Parámetro	Configuración	Comentarios
Plataforma	`OMAP3530`	Nombre del procesador empleado
Número de DSP's	`1`	Número de procesadores digitales de señal empleados
Interfaz física para DSP	`OMAP3530SHMEM`	Memoria compartida entre núcleos
Sistema operativo DSP	`DSPBIOS5XX`	Versión DSP/BIOS 5.41
Sistema operativo GPP	`OMAPLSP`	No empleado para este trabajo
Componentes DSPLink	`ponslrmc`	Selección de todos los componentes

Tabla 4 Configuración de DSPLink mediante el *script* en *perl*

Este procedimiento difiere con los presentados en los apartados anteriores (3.2.2.1.1 y 3.2.2.1.2) ya que ahora la herramienta proporciona información sobre la configuración del módulo previa a la generación de éste. Para ello, en primer lugar deberá ejecutarse el *script* escrito en *perl* denominado `dsplinkcfg.pl` en un terminal de consola del *host* Linux y en modo superusuario ya que el resultado de la misma genera una serie de ficheros que requerirá tener permisos de administrador. El *script* está ubicado en el directorio `/home/user/dsplink/config/bin/dsplinkcfg.pl` y su ejecución bloqueará el proceso hasta que el usuario no introduzca los parámetros de configuración del módulo.

[12] *Perl* es un lenguaje interpretado de propósito general originalmente desarrollado para la manipulación de texto y que en la actualidad se emplea para un amplio rango de tareas incluyendo administración de sistemas, desarrollo web o creación de *scripts* entre otras.

Para el caso de este trabajo, esos parámetros se corresponden con los presentados en la Tabla 4 y se escriben en el terminal de consola con una sintaxis particular, tal y como se muestra a continuación:

```
$ perl dsplink/config/bin/dsplinkcfg.pl --platform=OMAP3530 --nodsp=1
--dspcfg_0=OMAP3530SHMEM --dspos_0=DSPBIOS5XX --gppos=OMAPLSP --
comps=ponslrmc
```

De este modo, si alguna de las opciones configuradas no se hizo de forma correcta, se obtendrá un mensaje de error en el cual se muestran las posibles opciones correctas (ver Fig. 23). La fuente de error más habitual se debe a que cada parámetro va precedido de un guión doble y a continuación el nombre del parámetro, una sintaxis que en la primera toma de contacto puede llevar a cometer errores fácilmente. Finalmente, si la opción configurada es correcta, también se muestra una ventana indicando dicha configuración (ver Fig. 24).

```
******************** ERROR !!! ******************************

Please provide a valid Platform!
Following platform are supported currently:

ID-->DAVINCI
        DaVinci SoC - C64P DSP interfaced directly to ARM9
ID-->DAVINCIHD
        DaVinciHD SoC - C64P DSP interfaced directly to ARM9
        This platform does not supports multi DSP scenario
ID-->JACINTO1
        Jacinto SoC version 2 - C64P DSP interfaced directly to ARM9
        This platform does not supports multi DSP scenario
ID-->OMAP3530
        Omap3530 SoC - C64P DSP interfaced directly to ARM9
        This platform does not supports multi DSP scenario
ID-->DA8XX
        DA8XX SoC - C64P DSP interfaced directly to ARM9
        This platform does not supports multi DSP scenario
ID-->OMAPL1XX
        OMAP-L1XX SoC - C64P DSP interfaced directly to ARM9
        This platform does not supports multi DSP scenario
Provided:
Example: --platform=DAVINCI or --platform=<ID>
```

Fig. 23 Resultado de un parámetro de configuración de DSPLink erróneo

```
============================================================
Chosen configuration is as follows:

Chosen platform:
        Identifier:     OMAP3530
        Description:    Omap3530 SoC - C64P DSP interfaced directly to
ARM9
```

Fig. 24 Resultado de un parámetro de configuración de DSPLink correcto

Además de los configurados, hay una serie de parámetros extra en la configuración de DSPLink que son opcionales y que no han formado parte de la configuración elegida pero igualmente han sido analizados dada su útil funcionalidad en otras aplicaciones del mismo ámbito o en la propia aplicación desarrollada en este trabajo. Algunos de ellos permiten generar directorios temporales para almacenar tanto los ejecutables como las librerías de ARM y DSP, algo muy útil dada la compleja estructura de directorios de DSPLink; otra opción ofrece la posibilidad de que los subsistemas *hardware* tengan acceso directo a memoria (DMA) en vez de que la CPU copie cada porción de dato para las transferencias entre núcleos; también existen opciones de compatibilidad como la denominada *legacy support*, la cual garantiza el funcionamiento de aplicaciones escritas para versiones anteriores de DSPLink; y, por último, hay una opción de seguimiento (*trace*) útil para monitorizar y detectar errores en el funcionamiento de las aplicaciones.

3.2.2.1.4 Configuración de memoria a través de DSPLink

La siguiente fase en la configuración del módulo DSPLink involucra tanto a las secciones definidas del propio módulo como también a la configuración del módulo CMEM, ya que es necesario que DSPLink conozca su rango de direcciones para no generar errores en la ejecución de las aplicaciones. Así pues, la conformación de los rangos de memoria definidos para el núcleo 'ARM', deben coincidir con los que posteriormente se definan en el núcleo 'C64x+. La diferencia entre ambos radica en que mientras los cambios de memoria en el 'C64x+' pueden hacerse con independencia de sí el módulo *dsplinkk.ko* (y las librerías) ha sido generado o no, los cambios de memoria en el 'ARM' tienen que hacerse antes de la generación del módulo, ya que de lo contrario hay que recompilar éste, repitiendo nuevamente todo el proceso descrito hasta ahora y actualizando los componentes y librerías obtenidas de éste. Por tanto, es conveniente fijar una estructura de memoria, con sus secciones claramente definidas y una vez seguros de que la configuración es correcta, generar el módulo (y las liberías) y empezar a trabajar en la aplicación.

Las modificaciones de las secciones de memoria para el núcleo 'ARM' del procesador OMAP3530 se realizan en el fichero localizado en el siguiente directorio de la estructura de DSPLink: `$(DSPLINK)/config/all/CFG_OMAP3530_SHMEM.c`. Este fichero debe incluir los rangos de direcciones de memoria entre los cuales se encuentra cada sección definida en la introducción de este apartado así como la definición del módulo CMEM.

Un ejemplo de cómo se debe definir una nueva sección se muestra en la siguiente figura (ver Fig. 25) a través del propio fichero `CFG_OMAP3530_SHMEM.c`:

```
{
    10,                     /* ENTRY
    "CMEM",                 /* NAME
    0x87900000,             /* ADDRPHYS
    0x87900000,             /* ADDRDSPVIRT
    (Uint32) -1u,           /* ADDRGPPVIRT
    0x00500000,             /* SIZE
    TRUE,                   /* SHARED
    FALSE                   /* SYNCD
}
```

Fig. 25 Definición de una sección de memoria en el núcleo 'ARM' a través de DSPLink

En la imagen se muestran los diferentes campos que debe contener la definición de una sección, en concreto la sección perteneciente al módulo CMEM. En primer lugar, se define

un nuevo número de entrada y se le da un nombre indicativo a la sección. Posteriormente, se indican tanto la dirección física como la virtual, campos que deben ser iguales para que dicha sección pueda ser accesible por los dos núcleos que conforman el procesador OMAP3530. Después se escribe el tamaño deseado de la sección en hexadecimal y por último, a través del campo *shared,* se indica que se quiere compartir esa sección. Esta configuración se realiza para todas y cada una de las secciones de DSPLink citadas anteriormente (Vectores del Reset, DDR2, DSPLinkMEM, DSPLinkMEM1 y POOL_DSPLink) así como para el ejemplo mostrado.

Esta misma configuración debe hacerse también en el núcleo 'C64x+'. Sin embargo, tal y como se dijo anteriormente, no es necesario llevarla a cabo antes de la generación del módulo ya que es independiente de ésta. Por tanto, la configuración se realiza a través de archivos con extensión **.tcf,* los cuales permiten seleccionar diferentes características del sistema operativo DSP/BIOS como el mapa memoria, la planificación de tareas, la sincronización, la gestión de entrada/salida, etc. Para este Trabajo Fin de Máster ha sido necesario configurar las mismas secciones de memoria definidas para el núcleo 'ARM', las memorias caché y los registros de direcciones de memoria (MAR)[13] del procesador digital de señal[14]. Para ello, CCS proporciona un interfaz gráfico para configurar todas estas opciones de tal forma que el proceso se simplifica enormemente. El aspecto que presenta esa interfaz se muestra en la siguiente figura (ver Fig. 26):

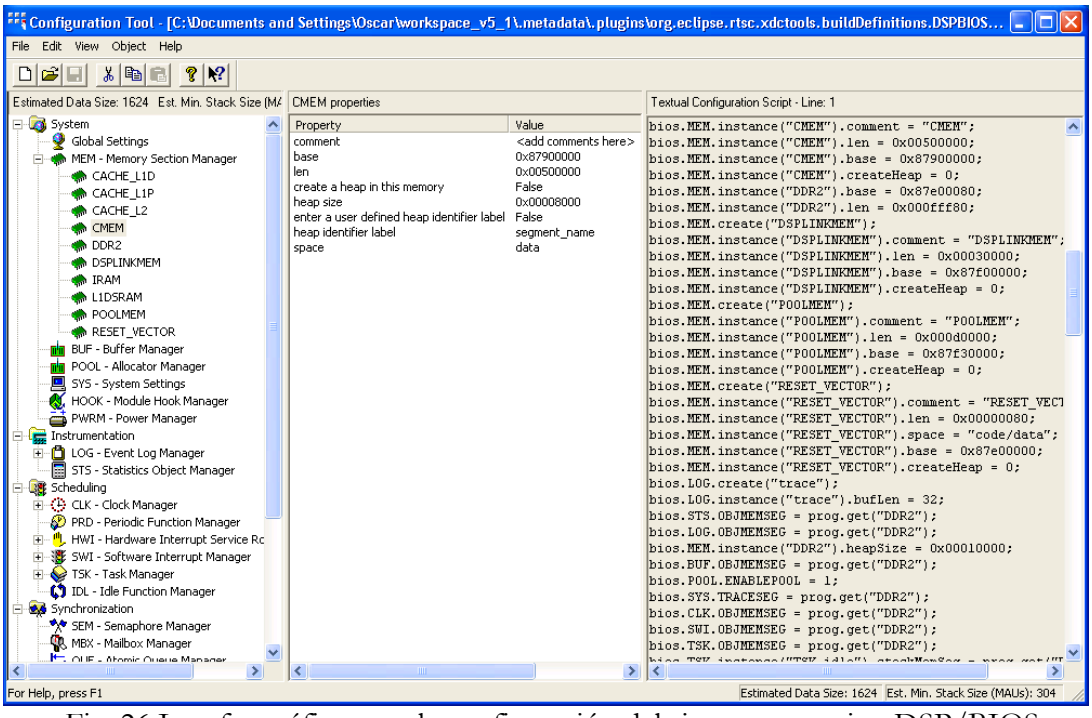

Fig. 26 Interfaz gráfico para la configuración del sistema operativo DSP/BIOS

En la Fig. 26 se muestran tres partes claramente diferenciadas: la situada más a la izquierda presenta las diferentes características del sistema operativo DSP/BIOS a las cuales se puede

[13] Los registros MAR (*Memory Attribure Register*) permiten la copia de datos a memoria caché dividiendo dicha memoria en secciones de 16 MB. Es imprescindible habilitarlos para el uso de la caché.

[14] La configuración completa del archivo es muy extensa y además se partía de una versión ya implementada en proyectos anteriores, por lo que si se desea consultar más información del fichero de configuración, acudir a [3].

acceder en este caso; la columna central representa las propiedades de la característica seleccionada anteriormente (la imagen muestra las diferentes propiedades del módulo CMEM); por último, la columna de la derecha es un *script* generado automáticamente por las acciones realizadas por el usuario en las otras dos columnas. Desafortunadamente, la versión CCS para Linux no implementa este entorno gráfico, por lo que es necesario en primer lugar realizar los cambios en un entorno Windows, posteriormente copiar la información del *script* (columna de la derecha) a un fichero de texto y por último, desde esa ubicación al fichero **.tcf* del entorno Linux.

3.2.2.1.5 Generación del módulo de DSPLink

Una vez completadas con éxito todas las fases anteriores, DSPLink ya tiene la información distribuida correctamente para poder generar su módulo. Para ello, nuevamente se vuelve a hacer uso de la utilidad *make*.

Hay que realizar un cambio de directorio, de manera que éste apunte a `$(DSPLINK)/gpp/src/` y ejecutar *make*[15]en modo superusuario Linux. Una vez que el proceso se complete de forma exitosa, el módulo *dsplinkk.ko* y las librerías de usuario son localizados en los siguientes directorios:

- `$(DSPLINK)/gpp/export/BIN/Linux/OMAP3530/DEBUG`
- `$(DSPLINK)/gpp/export/BIN/Linux/OMAP3530/RELEASE`

De forma análoga se realiza el mismo procedimiento en el DSP, de modo que el directorio para ejecutar *make* será `$(DSPLINK)/dsp/src/` y las librerías del DSP podrán encontrarse en los siguientes directorios:

- `$(DSPLINK)/dsp/export/BIN/DspBios/OMAP3530/OMAP3530_0/DEBUG`
- `$(DSPLINK)/dsp/export/BIN/DspBios/OMAP3530/OMAP3530_0/RELEASE`

Con esta última fase, da por finalizado el proceso de obtención del módulo *dsplinkk.ko*. La Fig. 27 muestra un diagrama de flujo sintetizado de las diferentes fases del proceso de configuración a modo aclaratorio en donde la parte izquierda de la imagen se corresponde con el tipo de usuario Linux (usuario, *user* o administrador, *root*) necesario para llevar a cabo cada fase; mientras que la parte derecha de la imagen indica los ficheros que son modificar para configurar correctamente el módulo.

[15] Existen opciones que acompañan al *make*, entre ellas, *make* –s que permiten visualizar el resultado de la compilación más compacto, sin mostrar las órdenes que son ejecutadas.

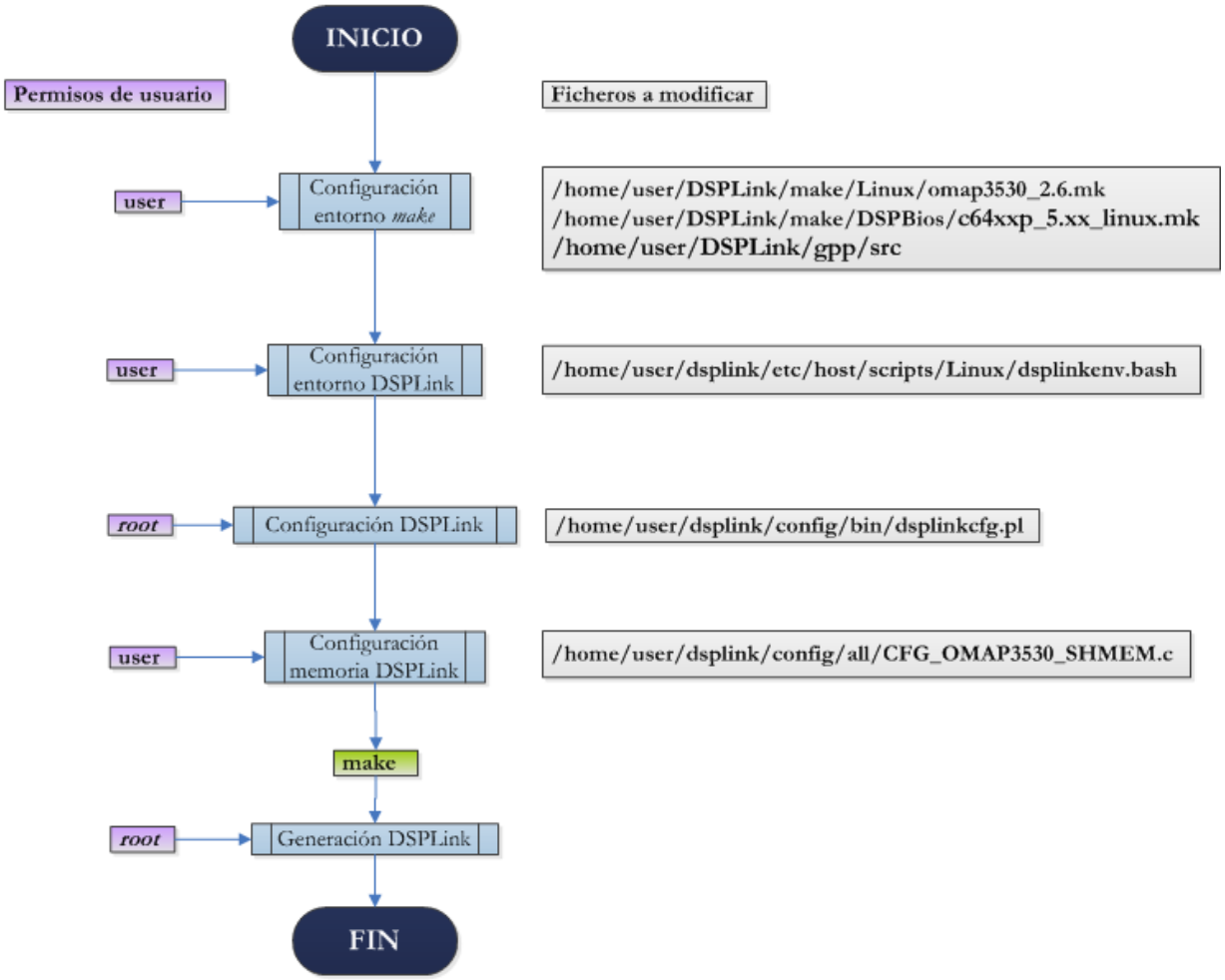

Fig. 27 Diagrama del proceso de configuración de DSPLink

Por último, antes de abordar cómo cargar el módulo recién generado en el *kernel* Linux embebido en el núcleo 'ARM', se va a describir el proceso de generación del módulo CMEM para completar la configuración del mapa de memoria del sistema embebido BeagleBoard.

3.2.2.2 CMEM

El módulo CMEM presenta un proceso de configuración mucho más sencillo que el recién analizado DSPLink. No incorpora componentes adicionales y sus opciones de configuración así como el conjunto de funciones que implementa son reducidas y bastante intuitivas.

CMEM permite definir secciones o lo que el fabricante denomina *pools* dentro del espacio de memoria que se le haya reservado, de tal forma que el espacio total asignado puede dividirse en hasta 128 *pools*. Esta característica permite separar las diferentes tareas en las que el módulo se vea involucrado, evitando sobrescrituras en la misma *pool*. Para este Trabajo Fin de Máster en concreto, se han definido dos *pools,* ya que se quiere separar el envío de la trama codificada y la recepción de imágenes descodificadas por parte del descodificador OpenSVC. Respecto a sus tamaños, para la primera de ellas se ha configurado un tamaño de 700 KB, valor más que suficiente ya que cada envío de

información del núcleo 'ARM' al 'C64x+' tiene un tamaño inferior a 100 KB, de modo que el valor escogido asegura el correcto funcionamiento de la aplicación. Por otro lado, el tamaño de la segunda *pool* está condicionado a su vez por el tamaño de la imagen descodificada. Como el monitor donde se visualizarán dichas imágenes presenta una resolución de 800 x 600 (480000 píxels) por el número de *bits* empleados por píxel, 16, se obtiene un valor mínimo cercano a 1MB. Este valor garantiza la descodificación de imágenes resolución PAL (720 x 576), más que suficiente para los objetivos de este Trabajo Fin de Máster. Por tanto, se ha optado por asignar 2 MB a la segunda *pool*, afianzando la copia de datos del núcleo 'C64x+' al 'ARM' y facilitando futuras ampliaciones de la aplicación desarrollada.

Esas asignaciones se realizan por medio de un *script* que se verá en el apartado 3.4 y su ejecución se lleva a cabo en el arranque del Linux embebido en el núcleo 'ARM'. El comando `$ cat /proc/cmem` permite visualizar la configuración descrita anteriormente[16] (ver Fig. 28).

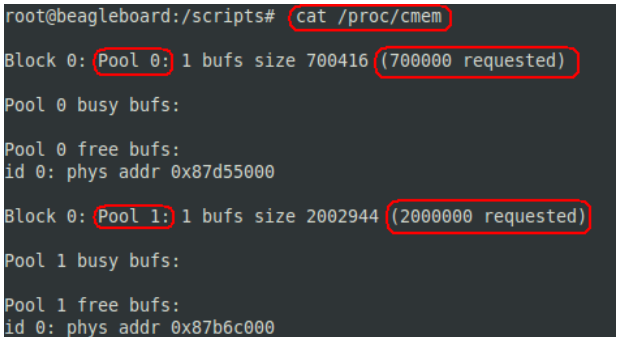

Fig. 28 Visualización de las *pools* de CMEM configuradas

En la imagen se muestran las *pools* demandadas, así como sus tamaños (se asigna el múltiplo más cercano a 1024) y sus respectivas direcciones físicas de inicio.

Una característica a tener en cuenta en la generación del módulo es el tamaño correspondiente a la memoria dinámica del módulo (*heap*) y el empleo de ella posteriormente. Esa memoria dinámica se obtiene como el resultado de restar el tamaño de cada *pool* creada al espacio total asignado a CMEM. En el caso de este trabajo la memoria dinámica tendría un valor cercano a los 2 MB, ya que se asignaron 5 MB al módulo CMEM, de los cuales casi 3 MB se han empleado para la reserva de *pools*.

Es necesario, tal y como recomienda Texas Instruments, que el tamaño de la *heap* sea suficiente para alojar los diferentes elementos creados de forma dinámica, ya que de lo contrario pueden ocurrir problemas de desbordamiento. De hecho, una de las funciones pertenecientes a la API de CMEM, denominada `CMEM_Alloc_Params`, permite ubicar dónde se van a almacenar los datos manejados por el módulo. Sin embargo, tal y como se verá en el apartado 4.3.2, será imprescindible para la aplicación desarrollada que los datos

[16] Para la comunicación con el sistema embebido BeagleBoard se emplea la interfaz serie RS-232 y el software Minicom, el cual proporciona un terminal de consola que permite gestionar los parámetros de la comunicación (*baudrate*, bits de datos, bit de parada, paridad, etc.) e interactuar tanto con el entorno de arranque *uboot* como con el propio sistema operativo embebido en el núcleo 'ARM' del procesador OMAP3530. Para más información acerca de la configuración acudir a [33].

compartidos entre los núcleos del procesador OMAP3530 se alojen en las *pools* reservadas y no en memoria dinámica.

Por tanto, teniendo estos importantes detalles de configuración en cuenta, se puede generar el módulo *cmemk.ko* correctamente. Se recuerda, como se mencionó en el apartado 2.3.2, emplear los ficheros fuente correspondientes al *kernel* Linux embebido en el proceso de creación del módulo, ya que de lo contrario se producirán errores de versiones entre los *kernels* empleados.

3.2.2.3 Carga de módulos en el *kernel* Linux embebido

La obtención de los módulos *dsplinkk.ko* y *cmemk.ko* correctamente es el penúltimo paso antes de poder trabajar con sus respectivas APIs en las aplicaciones desarrolladas. Por tanto, el último paso consiste en insertar los módulos recién generados en el *kernel* Linux embebido en el núcleo 'ARM' del procesador OMAP3530. Para ello, existen principalmente dos métodos: automático o manual.

El método automático permite insertar de forma transparente al usuario el módulo en el *kernel* en el arranque del sistema operativo. Para ello, es necesario copiar los módulos en el directorio `/lib/modules/2.6.32/kernel/drivers/dsp` perteneciente al sistema de ficheros de la tarjeta (distribución Angström) y a través de un *script* de configuración denominado *load_modules.sh* (ver apartado 3.4), realizar la carga de esos módulos. El método manual consiste en emplear el comando *insmod* y a continuación las órdenes para insertar el módulo correspondiente. La desventaja de este método respecto al primero es que cada vez que se tenga que reiniciar la tarjeta BeagleBoard, es necesario repetir el proceso, por lo que es claramente más eficiente la primera de las opciones vistas.

Una forma simple de comprobar si los módulos recién generados se han cargado correctamente en el *kernel* Linux es prestar atención al arranque del sistema operativo, ya que la última tarea que realiza antes de terminar dicho arranque es la carga de los módulos que se encuentran en el directorio indicado anteriormente (ver Fig. 29).

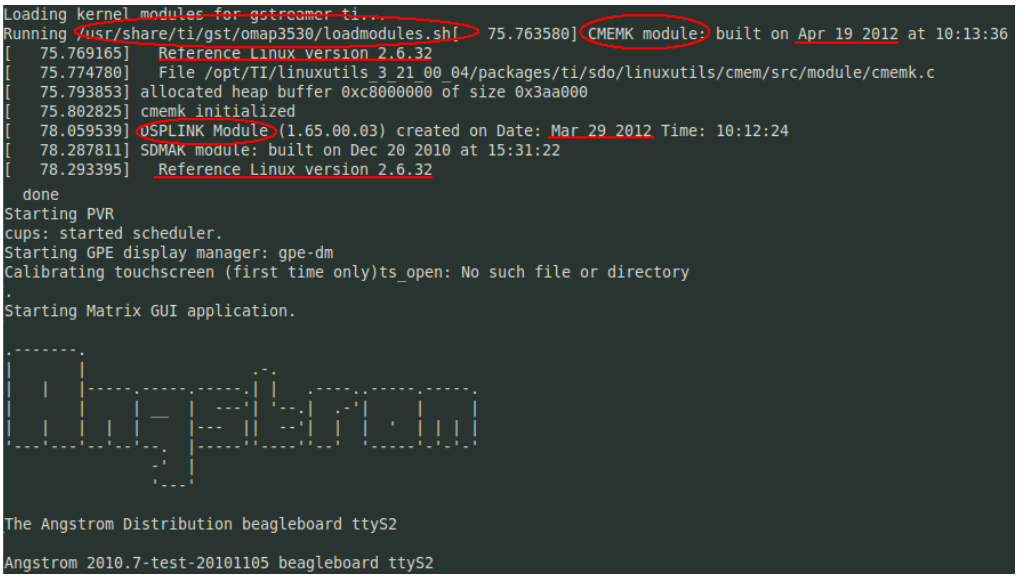

Fig. 29 Visualización de la carga automática de módulos en el *kernel* Linux embebido

En la imagen se han resaltado las acciones más significativas para la carga de módulos. En primer lugar, tal y como se mencionó líneas atrás, se ejecuta el *script* de configuración denominado *load_modules.sh*, que contiene las órdenes para insertar los módulos en el *kernel*. Después, la inserción de cada módulo lleva asociada la versión del *kernel* con el que fue generado[17] y la fecha y hora de su creación, convirtiéndose esta información en un método rápido de comprobar si realmente se está empleando el módulo generado por el usuario recientemente u otra versión desconocida. Si se trata de un módulo generado para una versión de *kernel* diferente, no se podrá emplear la API de ese módulo, generándose un error en la aplicación comunicando el error. Por último, destacar que uno de los mensajes pertenecientes al módulo CMEM es el correspondiente al tamaño de la *heap*, de modo que se puede corroborar si los tamaños de las *pools* creadas son coherentes con los definidos.

Con la carga de los módulos en el *kernel* Linux embebido, ya sí que se termina el proceso tanto de configuración del mapa de memoria del sistema embebido BeagleBoard como de gestión de módulos asociados al *kernel* Linux. El siguiente paso será describir cómo configurar el entorno de desarrollo CCS para editar y depurar las aplicaciones empleando, entre otras, las APIs de los módulos configurados.

3.3 Configuración del entorno de desarrollo Code Composer Studio

Descrita la parte de configuración del sistema operativo embebido Linux y su gestión de módulos, es el momento de describir cómo configurar el entorno a través del cual se construirán y depurarán las aplicaciones construidas para los dos núcleos del procesador OMAP3530. En este punto, el entorno de desarrollo integrado CCS se convierte en el elemento principal del sistema, ya que será la base fundamental sobre la que se apoyen dichas aplicaciones. En concreto, se van a diferenciar dos fases en el proceso de configuración del entorno:

- Adaptación y/o importación de proyectos[18] CCS [34].
- Depuración de ejecutables para núcleos 'ARM' y 'C64x+' [35].

En el primero de ellos se comentará el procedimiento empleado así como algunos inconvenientes encontrados mientras que la segunda de las fases supone una de las principales novedades respecto a proyectos anteriores realizados en el Grupo de Investigación GDEM y será analizada en profundidad, destacando por un lado el método de depuración empleado para cada núcleo y por otro lado, como a partir de esa descripción teórica, se lleva a cabo la configuración de esa depuración en CCS. En esta última parte se proporciona un ejemplo de configuración para una mejor comprensión de las operaciones realizadas.

[17] El propio usuario puede comprobar la versión del *kernel* que se ejecuta en el núcleo 'ARM' del procesador OMAP3530 ejecutando el comando `uname -r` una vez completado el arranque del sistema operativo embebido.

[18] Se hace referencia a proyecto como el conjunto de ficheros, tanto ficheros fuente como de configuración, que permite la obtención de un ejecutable tras la acción de la *toolchain* correspondiente.

3.3.1 Adaptación y/o importación de proyectos CCS

Tanto para el núcleo 'ARM' como para el 'C64x+' se ha comenzado a trabajar con proyectos que se encontraban funcionando, bien en otras máquinas PC o bien en versiones anteriores a CCS, por lo que *a priori* no sería necesario realizar numerosas modificaciones en los proyectos de ambos núcleos. Sin embargo, han surgido algunos inconvenientes asociados a la compleja estructura del proyecto o a la heterogeneidad de versiones entre sistemas operativos diferentes que han requerido cambios importantes.

En este Trabajo Fin de Máster, se han empleado principalmente dos tipos de proyectos, uno para cada núcleo del procesador OMAP3530: para el núcleo 'ARM' se ha trabajado sobre un proyecto *makefile,* donde CCS solicita un directorio con ficheros fuente y un fichero *makefile*. Ambas premisas las cumple el proyecto construido para el reproductor multimedia MPlayer. Sin embargo, no fue posible emplear la herramienta de Texas Instruments para realizar las tareas de compilación y enlazado dada la complejidad del proyecto y el elevado número de ficheros y librerías que contiene éste. En consecuencia, se emplea CCS para editar los archivos pertenecientes al reproductor a modo de interfaz gráfica, pero los procesos relativos a la obtención del ejecutable se realizan en un terminal de consola de Linux mediante la utilidad *make*.

Por otro lado, para el núcleo 'C64x+' se ha empleado la funcionalidad de importar proyectos que incorpora CCS, manteniendo en principio la compatibilidad con versiones anteriores de proyectos. En concreto, se han importado proyectos CCS de la versión 3.3 operando en el sistema operativo Windows a la versión 5.1 trabajando en Linux salvando los inconvenientes que conlleva esta importación como por ejemplo la actualización de algunos directorios que apuntaban a herramientas empleadas o las dependencias entre los subproyectos que componen el proyecto principal. A diferencia de lo que sucedía para el núcleo 'ARM', ahora sí que es posible emplear la *toolchain* para el procesador digital de señal a través de CCS, por lo que se editan los ficheros y se lanza el proceso de generación del ejecutable en la misma interfaz gráfica.

La Fig. 30 muestra los dos tipos de proyectos comentados para los núcleos 'ARM' y 'C64x+' respectivamente:

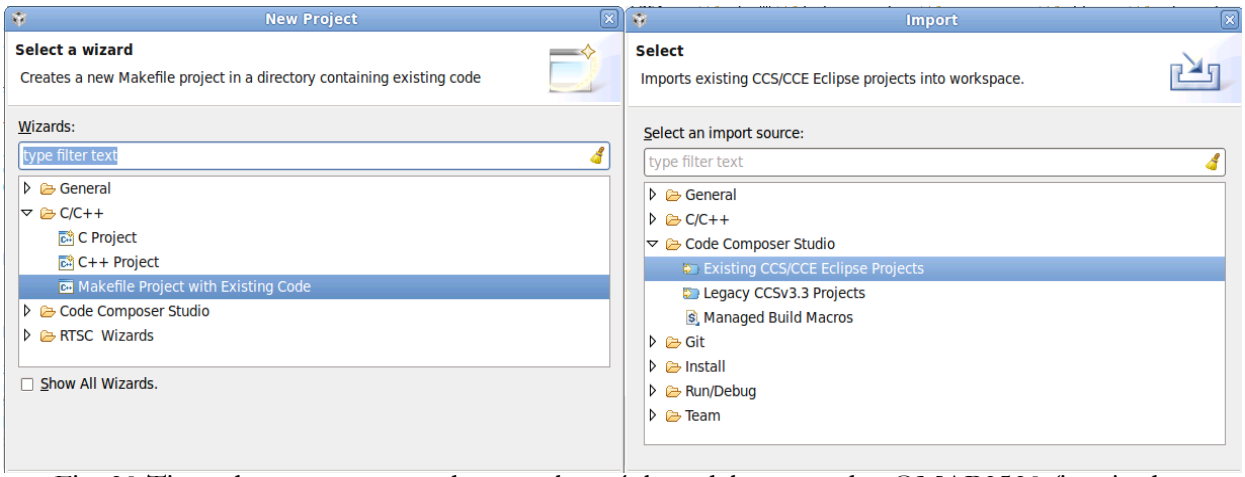

Fig. 30 Tipos de proyectos creados para los núcleos del procesador OMAP3530 (izquierda: 'ARM'; derecha: 'C64x+')

Algunas diferencias y similitudes entre los dos tipos de proyectos se comentan a continuación: en primer lugar, resaltar el elevado tiempo de compilación y enlazado hasta la obtención del ejecutable de ambos proyectos debido al elevado número de ficheros de diverso tipo que contienen (librerías, ficheros fuente, de cabecera, de configuración, *makefiles*, etc.). Esto conlleva a que cualquier mínimo cambio en algún fichero del proyecto construido demore el resultado final, convirtiéndose éste en uno de los principales *hándicaps* del trabajo desarrollado. En cuanto a las diferencias, aparte de la ya comentada anteriormente acerca del uso de la *toolchain* a través de CCS, existe una particularidad extra: en el caso del proyecto construido para el núcleo 'ARM', toda la estructura de ficheros de MPlayer no está físicamente en el directorio de trabajo (*workspace*) junto al resto de proyectos, sino que está redireccionada desde una carpeta del sistema de ficheros del *host* Linux. Esto se debe al tipo de proyecto empleado, ya que en el caso de los proyectos del núcleo 'C64x+', sí que se dispone de ellos en el *workspace*. Esta diferencia es importante tenerla en cuenta si el usuario del sistema decide eliminar el proyecto de MPlayer del *workspace,* ya que no estará borrando el proyecto del lugar de trabajo sino del propio sistema de ficheros, eliminando por completo el trabajo desarrollado hasta ese momento.

Además de lo comentado hasta ahora, hay que tener en cuenta las típicas pero necesarias operaciones de inclusión de directorios que contienen ficheros cabecera, librerías, etc. para evitar errores en las fases de compilación y enlazado del proyecto. No han sido descritas al tratarse de tareas clásicas que en ocasiones son muy molestas pero que en definitiva son un paso irrelevante respecto al objetivo principal de este Trabajo Fin de Máster.

3.3.2 Depuración de ejecutables para núcleos 'ARM' y 'C64x+'.

Una vez que se han obtenido los ejecutables pertenecientes a los dos núcleos del procesador OMAP3530 mediante los procedimientos descritos en el apartado anterior, se van a describir la estructura y configuración de la depuración para ellos. Antes de comenzar, es importante destacar que CCS permite definir dos tipos de depuración principalmente:

- *Software:* la aplicación se depura de forma remota mediante alguna herramienta extra que proporcione esa funcionalidad.
- *Hardware:* la aplicación se depura a través de un emulador o elemento *hardware* que permita realizar la comunicación físicamente.
- *Launch Group*: esta utilidad permite lanzar varias sesiones de depuración simultáneamente, independientemente del tipo que sean.

Dada la metodología de trabajo seguida en el GDEM, se realizan depuraciones diferentes para los núcleos 'ARM' y 'C64x+'. Por un lado, se emplea nuevamente la herramienta *software* Codesourcery para depurar de forma remota las aplicaciones construidas para el núcleo 'ARM', en concreto la utilidad *gdb* (GNU Debugger), mientras que por otro lado se utilizan los ya mencionados emuladores *hardware* de Blackhawk para depurar las aplicaciones del núcleo 'C64x+'. Esto es posible ya que CCS soporta ambos modos de depuración y especialmente la versión 5.1 soporta un modo más de gran utilidad para

abordar el objetivo de este Trabajo Fin de Mäster que permitirá realizar depuraciones multi-procesador y que más adelante se comentará.

A continuación, se detallará el procedimiento para configurar ambos modos de depuración, analizando en ambos casos en primer lugar la estructura (ver Fig. 31 para 'ARM' y Fig. 35 para 'C64x+') y posteriormente la configuración a través de CCS.

3.3.2.1 Estructura y configuración de la depuración *software* – núcleo 'ARM'

La utilidad *gdb* [36] del paquete Codesourcery incluye un servicio de depuración remota a través de un enlace serie o una conexión TCP/IP, permitiendo establecer una sesión de depuración sobre el núcleo 'ARM' del procesador OMAP3530 con soporte para puntos de ruptura, ejecución paso a paso y otras funciones de control. El sistema presenta un servidor de depuración ubicado en el sistema embebido BeagleBoard (servidor GDB) y un cliente de depuración situado en el *host* Linux. El ejecutable deberá tener símbolos de depuración integrados (opción –g del compilador), y existirá tanto en el sistema de ficheros de la tarjeta BeagleBoard como en el del *host* Linux. Además, es necesario que la utilidad *gdb* instalada en el cliente conozca también los ficheros fuente de la aplicación para que se puedan visualizar el flujo y comportamiento del programa construido depurando paso a paso y colocando puntos de ruptura. Para este Trabajo Fin de Máster se ha empleado el protocolo TCP/IP en vez del enlace serie para comunicar cliente y servidor al tener implementado el protocolo NFS en el sistema.

La estructura de la depuración descrita anteriormente se muestra en la siguiente figura (ver Fig. 31):

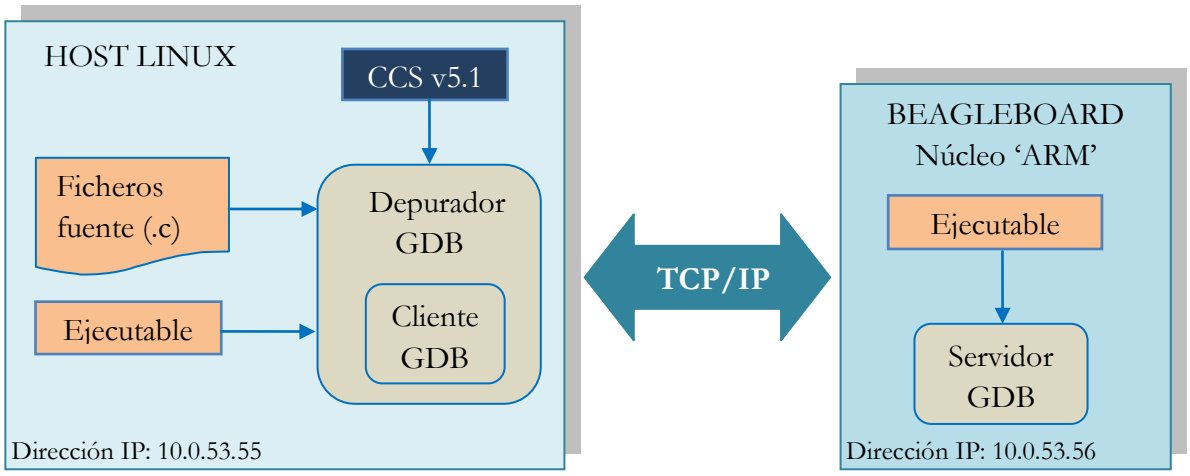

Fig. 31 Estructura de depuración para el núcleo 'ARM'

El procedimiento acerca de cómo configurar el cliente de depuración a través de CCS y algunas particularidades necesarias a realizar en el núcleo 'ARM' para que la depuración se lleve a cabo correcta y ordenamente se van a describir en este punto. Para ello, se ha optado por emplear dos imágenes (ver Fig. 32 y Fig. 33) que muestran de forma gráfica las configuraciones realizadas en CCS a modo de ejemplo y que servirán de apoyo a la descripción de cada uno de los campos contenidos en dichas imágenes, destacando el nombre del campo y su contenido de aquellos más relevantes.

En primer lugar, hay que crear una nueva sesión de depuración *software* denominada *C/C++ Remote Application* (ver Fig. 32, parte izquierda). Una vez hecho esto, el entorno de desarrollo permite configurar una serie de campos a través de diferentes pestañas: *main, debugger, source y common* (ver Fig. 32, parte superior). En concreto, para este trabajo ha sido necesario modificar los campos contenidos bajo las pestañas *main* y *debugger*, de manera que a continuación se indicará el significado de esos campos y un valor a modo de ejemplo tomando como referencia los contenidos de las imágenes citadas anteriormente (Fig. 32 y Fig. 33).

- Pestaña *main* (ver Fig. 32):
  - Configuración del tipo de depuración empleada que satisfaga la estructura cliente-servidor descrita para el núcleo 'ARM' (ver Fig. 32, ventana inferior): `GDB (DSF) Manual Remote Debugging Launcher.`
  - Nombre del ejecutable construido para 'ARM': `/home/user/dsplink/gpp/export/BIN/Linux/OMAP3530/DEBUG/ARM_binary_ADA.`
  - Nombre del proyecto de la aplicación en CCS: `ApplicationARM.`
  - El resto de opciones pertenecientes a esta pestaña y englobadas bajo el campo *Build (if required) before launching* se han mantenido por defecto, tal y como muestra la Fig. 32 al no afectar a la configuración realizada.

- Pestaña *debugger*, opción *main* (ver Fig. 33 , parte superior):
  - Ubicación dentro del sistema de ficheros del *host* de la utilidad *gdb* de Codesourcery: `/opt/codesourcery/arm-none-gnueabi/arm-2010.09/bin/arm-none-gnueabi-2010.09-gdb.`
  - Fichero de comandos con funciones de *time-out* y librerías necesarios para la depuración: `/home/user/workspaceCCSv5/gdb-ini.txt.`
  - El resto de opciones pertenecientes a la opción *main* se han dejado por defecto al no afectar a la configuración realizada.

Respecto al fichero de comandos mencionado, el procedimiento habitual es que primero se arranque el servidor y luego se lance la petición de conexión desde CCS. La función time-out del fichero contiene un primer comando que permite al cliente de depuración *gdb* esperar sin límite de tiempo una respuesta por parte del servidor *gdb*. Por otro lado, el segundo comando presenta más funcionalidad respecto al primero, ya que su función es la de indicar a *gdb* donde se encuentran las librerías de la herramienta Codesourcery, proceso necesario para no generar errores en la depuración.

- Pestaña *debugger*, opción *connection* (ver Fig. 33 , parte inferior):
  - Tipo de conexión entre cliente y servidor: `TCP.`
  - Dirección IP del servidor gdb (sistema embebido BeagleBoard): 10.0.53.56.
  - Número de puerto para establecer la comunicación entre cliente y servidor: `10000.`

Una vez configuradas estas opciones, el cliente de depuración ya está preparado para comenzar la sesión de depuración. A continuación se presentan las dos imágenes a las que se ha hecho referencia (ver Fig. 32 y Fig. 33) para corroborar el contenido de los campos presentados..

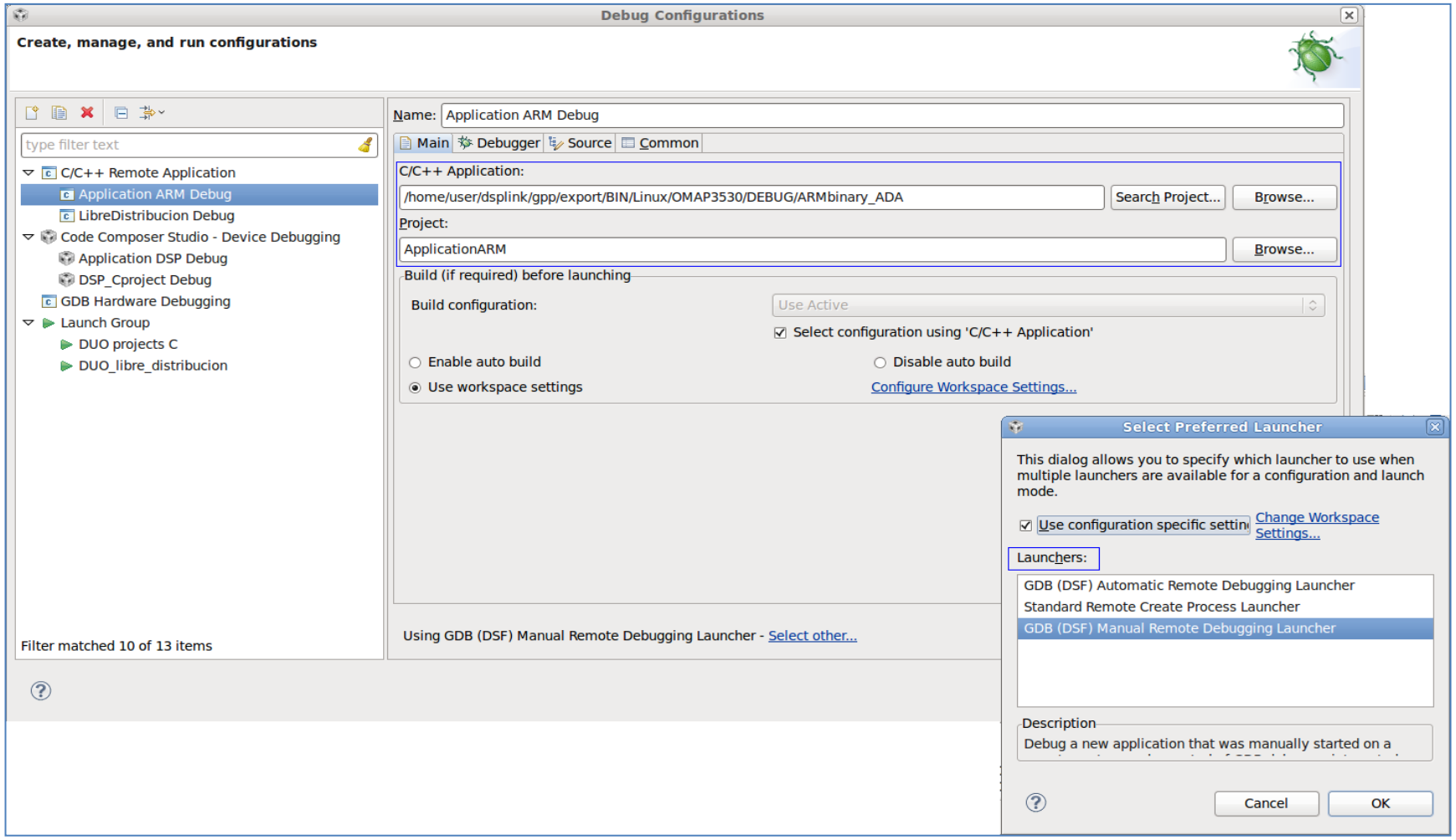

Fig. 32 Configuración de la depuración en Code Composer Studio para el núcleo 'ARM' – pestaña *main*

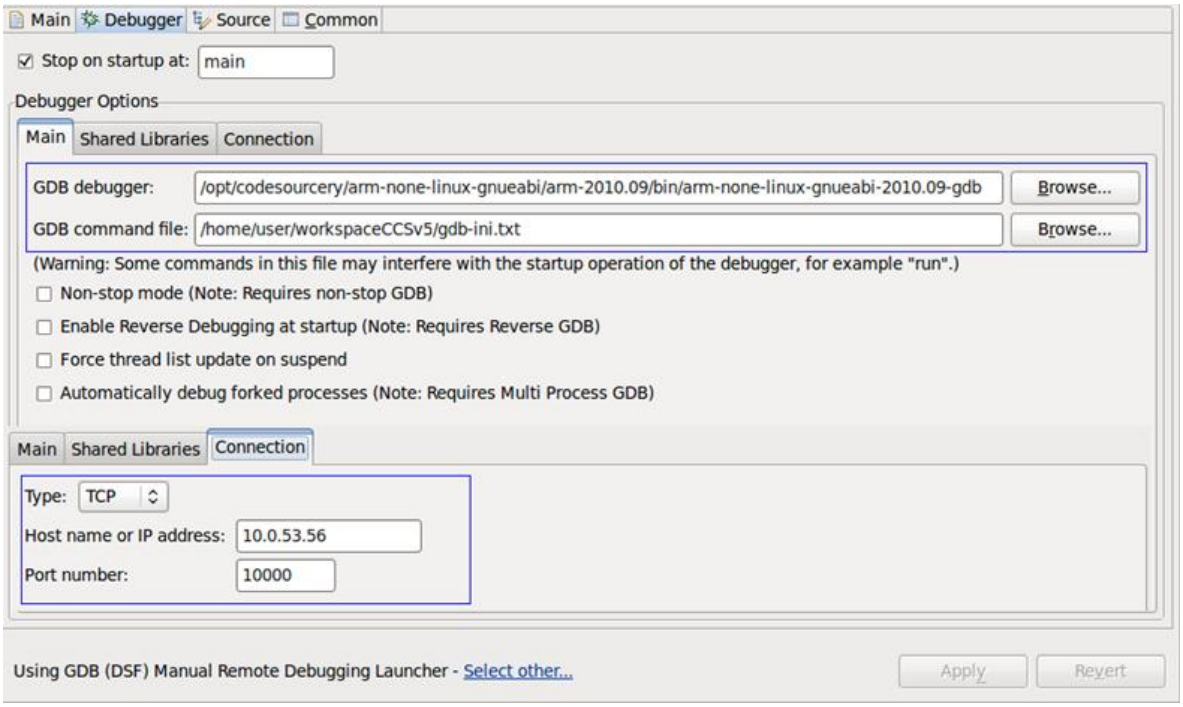

Fig. 33 Configuración de la depuración en Code Composer Studio para el núcleo ARM – pestañas *debugger* y *connection*

Respecto al servidor de la depuración, es necesario disponer, tal y como se dijo, de la utilidad *gdb* de CodeSourcery en el sistema de ficheros embebido en el núcleo 'ARM', para posteriormente lanzar dicho servidor a través de la terminal de consola que permite interactuar con el sistema operativo. Para realizar estas operaciones se ha elaborado un *script* denominado *mplayer_execution.sh* (ver apartado 3.4) que contiene los siguientes campos[19]:

- Ubicación del servidor en el sistema de ficheros del núcleo 'ARM'.
- Ubicación del enlazador dinámico *ld-linux.so.3*, encargado de cargar en memoria las diferentes librerías de usuario del sistema que la aplicación necesite usar en tiempo de ejecución. Esas librerías se localizan mediante la opción *--library-path*.
- Puerto de comunicaciones (configurado el 10000 en la descripción realizada) y nombre del ejecutable a depurar.

Por consiguiente, si se ha realizado correctamente la configuración, cuando se arranque el servidor *gdb* en el sistema de ficheros Linux embebido a través del *script* y se lance la depuración en el cliente a través de CCS, la terminal de consola que interactua con el sistema operativo debe proporcionar los siguientes mensajes (ver Fig. 34):

```
root@beagleboard:/scripts# mplayer execution.sh
Arranque del servidor GDB en la tarjeta
Process ./ARMbinary ADA created; pid = 1225
Listening on port 10000
Remote debugging from host 10.0.53.55
```

Fig. 34 Ejecución del servidor *gdb* mediante *script* de configuración *mplayer_execution.sh*

[19] Para visualizar el contenido de los campos descritos a continuación, acudir al apartado 3.4 del documento.

En la imagen se muestra la escucha del servidor *gdb* por el puerto configurado (`10000`) y posteriormente, una vez que se ha establecido la comunicación, es decir, que se ha iniciado la depuración a través de CCS, se informa al usuario que se está depurando de forma remota la aplicación cliente (`ARMbinary_ADA`) ubicada en la dirección IP 10.0.53.55 que corresponde al *host* Linux.

3.3.2.2 Estructura y configuración de la depuración *hardware* – núcleo 'C64x+

En el caso del 'C64x+', la técnica empleada difiere por completo de la vista en el 'ARM'. Se trata de una depuración más clásica, mediante un emulador *hardware* del fabricante Blackhawk modelo USB560M a través de la interfaz JTAG. La estructura empleada para este núcleo del procesador OMAP3530 se muestra en la Fig. 35:

Fig. 35 Estructura de depuración para el núcleo 'C64x+'

Para llevar a cabo este tipo de depuración en el entorno de desarrollo integrado CCS se requieren dos configuraciones:

- Configuración de un *target* (archivo con extensión **ccxml*).
- Configuración de la propia depuración.

La primera de ellas consiste en añadir un fichero al proyecto para el núcleo 'C64x+' en el que se le indiquen la tarjeta de desarrollo o sistema embebido y la conexión empleada para comunicarse con la primera. Para este Trabajo Fin de Máster, la configuración se muestra en la siguiente tabla (ver Tabla 5):

PARÁMETRO	Valor
Sistema embebido	`OMAP3530`
Conexión	`Blackhawk USB560-M Emulator`

Tabla 5 Configuración del *target* para el núcleo 'C64x+'

Además, mediante la configuración avanzada del *target*, hay que modificar los campos en los que se ofrece la posibilidad de emplear ficheros GEL para los distintos procesadores del dispositivo seleccionado [37]. Estos ficheros se encargan, entre otras tareas, de inicializar automáticamente el dispositivo, reconocer que la tarjeta se encuentra conectada o compilar y cargar el programa al iniciar CCS. Estas operaciones no interesan para el objetivo de este trabajo ya que, en aplicaciones que comuniquen ambos núcleos del procesador OMPA3530, será el núcleo 'ARM' el que se encargue de inicializar al núcleo 'C64x+' en el

momento adecuado. En consecuencia, habrá que dejar vacío el campo correspondiente al fichero GEL para el 'C64x+' (*Initialization script*). Asociado a esto, el *target* muestra además el resto de procesadores del dispositivo seleccionado (*Cortex_A8_0*, *cs_child_0*, *cs_child_1*, etc.). Como en esta ocasión solo interesa emplear el 'C64x+', el resto de dispositivos serán descartados marcando para cada uno de ellos la opción *bypass* dentro de la propia configuración avanzada del fichero. Estas acciones descritas se muestran en la siguiente figura (ver Fig. 36).

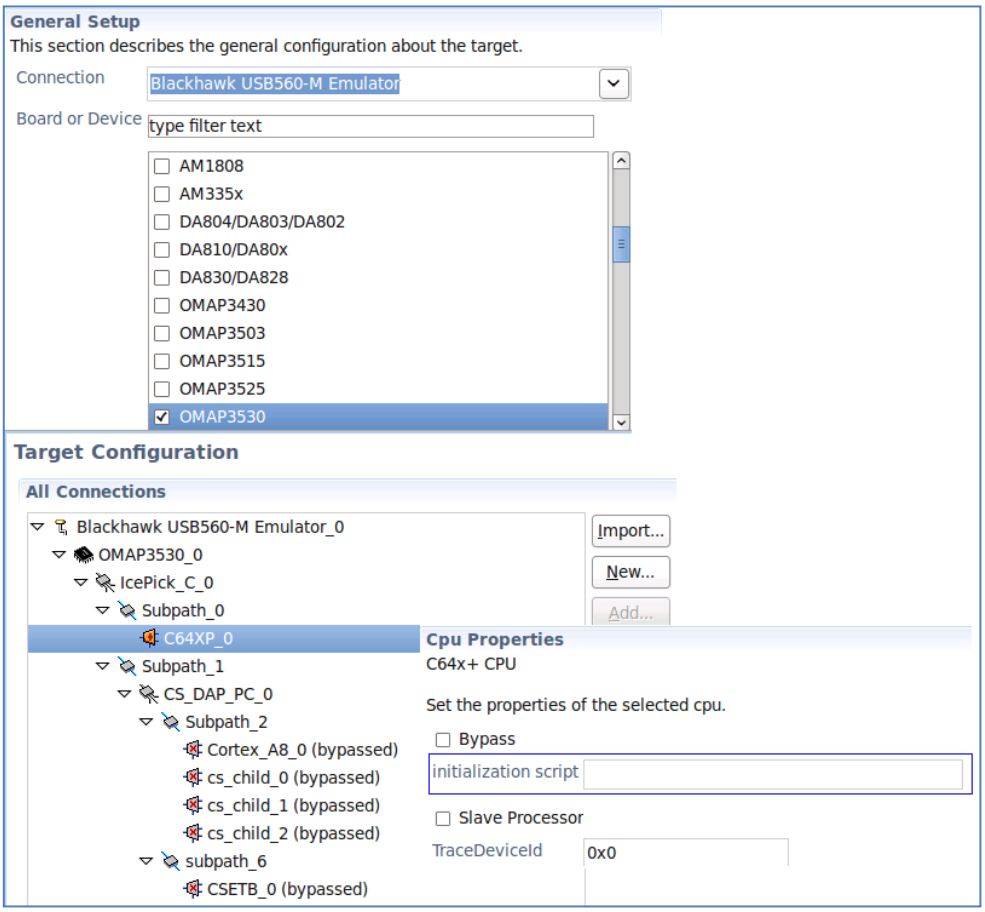

Fig. 36 Configuración del *target* en Code Composer Studio para el núcleo 'C64x+'

A continuación, al igual que se hizo para el procesador de propósito general del procesador OMAP3530, se va a describir las diferentes acciones a realizar en CCS para depurar con éxito aplicaciones para el núcleo 'C64x+'. Para comenzar, en este caso hay que crear una nueva sesión de depuración *hardware*, denominada *Code Composer Studio Device Debugging* (ver Fig. 37, parte izquierda). Una vez seleccionado el tipo de depuración, se van a describir los campos modificados para la realización de este trabajo englobado bajo las pestañas *main, program* y *target*, así como su contenido a modo de ejemplo, tomando como referencia los de la Fig. 37.

- Pestaña *main* (ver Fig. 37, parte superior)*:*

  - Directorio completo del *target* configurado: `/home/user/ServidorNFS/nfs02/workspaceCCSv5.1/ApplicationDSP/DSPemulator.ccxml`.

- Adición/Eliminación de ficheros GEL de inicialización: `Sin script de inicialización`.

Bajo la configuración de estos campos, se encuentran las CPUs disponibles para la sesión de depuración creada. En este caso se visualiza únicamente el núcleo 'C64x+' ya que el resto de dispositivos fueron descartados mediante la opción *bypass* anteriormente comentada (ver Fig. 36). Si alguno de ellos fuera seleccionado (desmarcar la opción *bypass*), aparecería junto al 'C64x+'.

- Pestaña *program* (ver Fig. 37, parte intermedia)*:*

  - Selección del dispositivo a emplear: `Blackhawk USB560-M Emulator_0/ C64XP_0`.
  - Nombre del proyecto: `ApplicationDSP`.
  - Nombre del ejecutable construido para 'C64x+': `/home/user/dsplink/dsp/export/BIN/DspBios/OMAP3530/OMAP3530_0/DEBUG/ada_dsp.out`.
  - Configuración de las opciones de carga (programa/símbolos): `solo símbolos`.

La configuración de las opciones de carga permite, una vez lanzada la sesión de depuración del núcleo 'C64x+' y realizada la conexión con el núcleo 'ARM', cargar el programa o cargar únicamente los símbolos de depuración. Como se ha comentado con anterioridad, en aplicaciones donde interactúan ambos núcleos del procesador OMAP3530, el 'ARM' es el que tiene el control sobre el 'C64x+', de modo que será el primero el encargado de cargar la aplicación del núcleo 'C64x+' en este procesador. Por esta razón, la opción adecuada para el sistema embebido BeagleBoard será la de cargar solo los símbolos de depuración[20].

- Pestaña *target* (ver Fig. 37, parte inferior):

  - Selección del dispositivo a emplear: `Blackhawk USB560-M Emulator_0/ C64XP_0`.
  - Configuración de opciones de depuración (conexión al inicio, reiniciar al conectar con la plataforma, etc.).

Es necesario configurar algunas opciones del depurador asociadas al arranque del mismo como no seleccionar la opción que conecta el núcleo 'C64x+' al inicio de la depuración (denominada en CCS *Connect to the target on debugger startup*), ya que éste se encuentra en estado de *reset* y ralentizará el proceso de depuración intentándolo conectar; o también no marcar la inicialización del núcleo 'C64x+' al conectar (denominada en CCS *Reset the target on a connect*) ya que es el 'ARM' quien tiene el control de esas acciones. Se ofrece la siguiente figura (ver Fig. 37) con los campos mencionados detallados para una mejor comprensión de esta configuración.

[20] Para poder cargar únicamente los símbolos de depuración, el compilador del núcleo 'C64x+' debe tener habilitada la opción –g que permite generar información de depuración al ejecutable correspondiente.

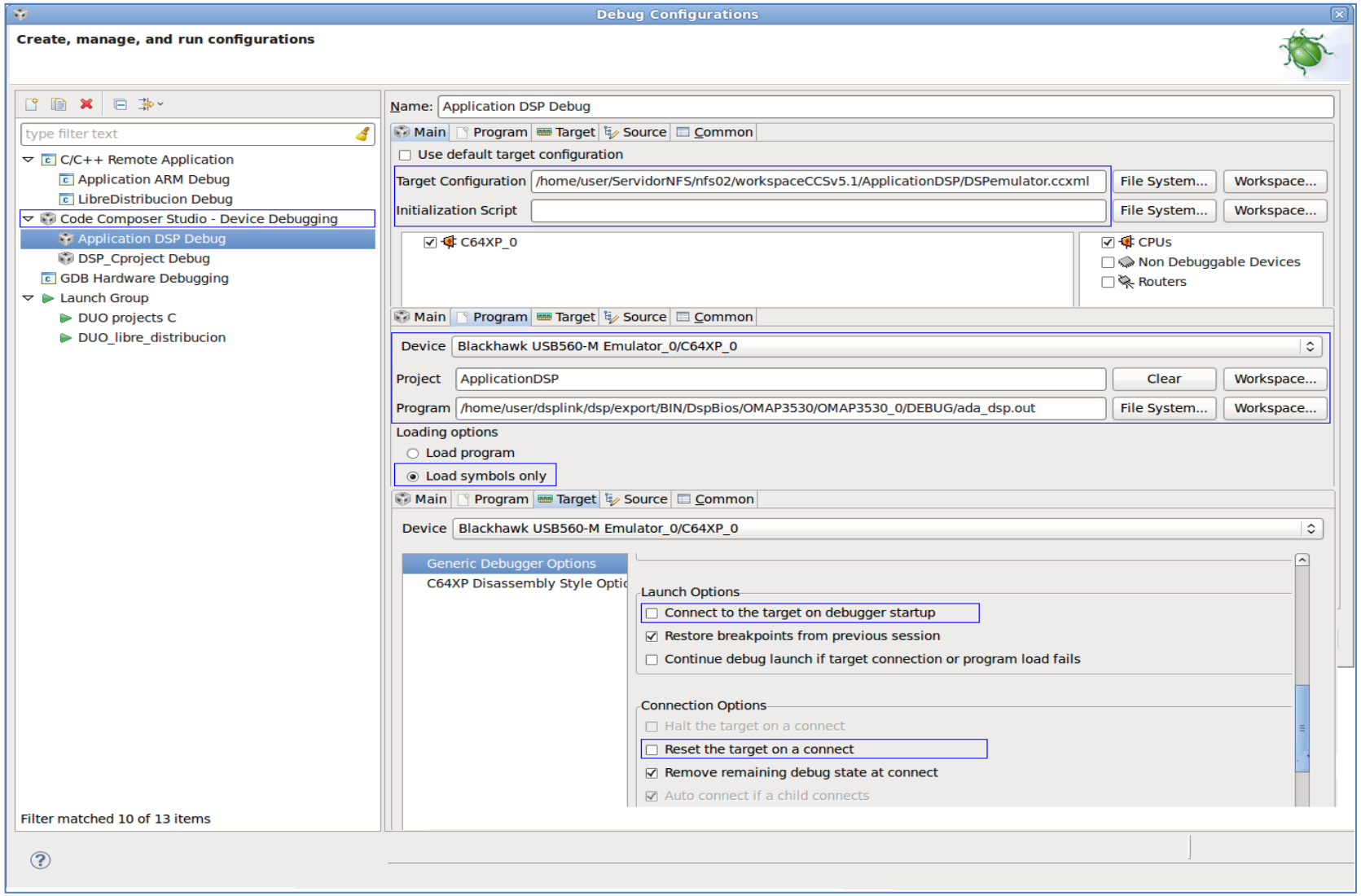

Fig. 37 Configuración de la depuración en Code Composer Studio para el núcleo 'C64x+'

3.3.3 Depuración multiprocesador

Una vez vistas de forma individual las dos configuraciones realizadas para los núcleos 'ARM' y 'C64x+', se va a presentar una de las principales novedades que ha introducido la versión 5.1 de CCS en cuanto a modos de depuración y que ha supuesto no solo una gran ventaja en la realización de este Trabajo Fin de Máster sino también un cambio en la metodología que se venía realizando en el GDEM en proyectos anteriores. La nueva versión de CCS permite lanzar simultáneamente diferentes configuraciones de depuración independientes, con el fin de poder depurar aplicaciones multi-procesador. El objetivo dentro de este trabajo es crear una de estas configuraciones donde añadir los dos tipos de configuración analizados (depuración *software* para el núcleo 'ARM' y *hardware* para el núcleo 'C64x+'). El aspecto que presenta esta novedosa característica de CCS se muestra en la siguiente figura (ver Fig. 38):

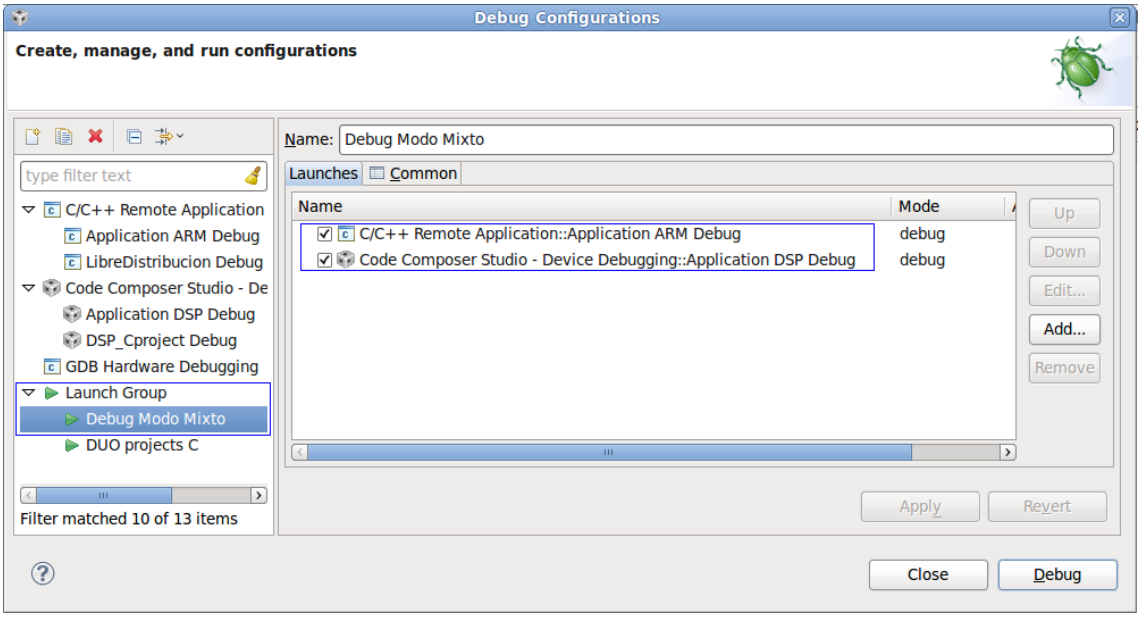

Fig. 38 Configuración de la depuración multiprocesador en Code Composer Studio

Para la configuración de este tipo de depuración, hay que crear dentro de la sección *Launch Group* una nueva sesión de depuración a la que se deben añadir las depuraciones de los núcleos del procesador OMAP3530. Es importante resaltar en este punto la opción que se comentó anteriormente de conectar el núcleo 'C64x+' inmediatamente después de lanzar la depuración, ya que es a partir de este momento cuando es posible conectar el 'C64x+' con la aplicación. Se aconseja no marcar esa opción, tal y como se indicó antes, para evitar molestos mensajes que notifican que el 'C64x+' no ha salido del estado inicial (*reset*). Esta operación la realizará el procesador de propósito general cuando la aplicación así lo requiera. La Fig. 39 muestra un determinado momento de una depuración en la cual los dos núcleos del OMAP3530 han establecido la comunicación correctamente e interactúan entre ellos.

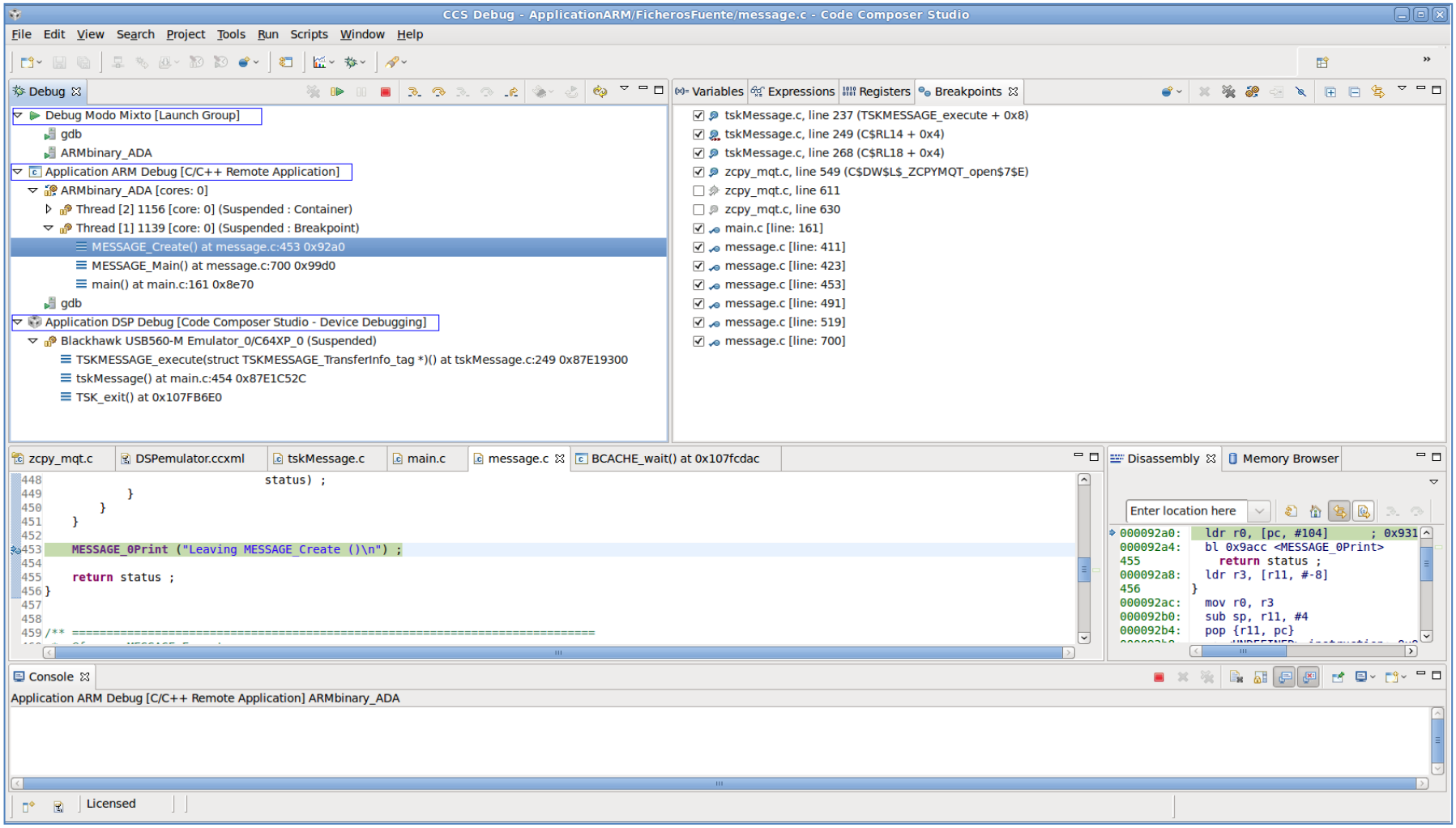

Fig. 39 Perspectiva de depuración multiprocesador en Code Composer Studio

En la imagen, en concreto en la pestaña *debug*, se visualiza el árbol de la estructura de la depuración: el nivel alto se corresponde con el nombre asignado al grupo de depuración, llamado *Debug Modo Mixto*; a continuación se presentan las dos configuraciones individuales que forman dicho grupo: *Application ARM Debug* y *Application DSP Debug* que pertenecen a los núcleos 'ARM' y 'C64x+' respectivamente. Por último, en el nivel más bajo de la estructura se encuentran los ficheros fuente y de cabecera que componen los diferentes proyectos construidos.

Para realizar con éxito la comunicación con el núcleo 'C64x+', el procedimiento seguido se ha basado en el que se venía realizando en el GDEM. Se describe en los siguientes puntos:

1. Inclusión de las funciones del módulo DSPLink que permiten establecer la comunicación entre los núcleos del procesador OMAP3530. En los apartados 4.3.1y 4.3.5 se analizarán qué funciones y por qué se ha elegido esa ubicación.
2. Ejecución de una serie de funciones de DSPLink (`PROC_Setup`, `PROC_Attach`, `PROC_load` y `PROC_Start`) [38] en el núcleo 'ARM' que realizan el enlace, la carga y el inicio del programa en el núcleo 'C64x+'.
3. Detección de la ejecución en el núcleo 'ARM' y conexión del otro núcleo del sistema embebido BeagleBoard, ya que ahora sí ha salido del estado *reset*.
4. Carga únicamente de los símbolos de depuración del ejecutable del núcleo 'C64x+', proceso que se puede realizar en la configuración de la depuración o ya una vez lanzada ésta.
5. Comunicación establecida entre los núcleos del procesador OMAP3530 de manera que ambos se encuentran listos para realizar operaciones que los involucren en la misma aplicación final.

Cuando ya se tiene claro el procedimiento inicial para comunicar ambos núcleos, la colocación de puntos de ruptura acelera este proceso. La Fig. 40 representa el procedimiento descrito.

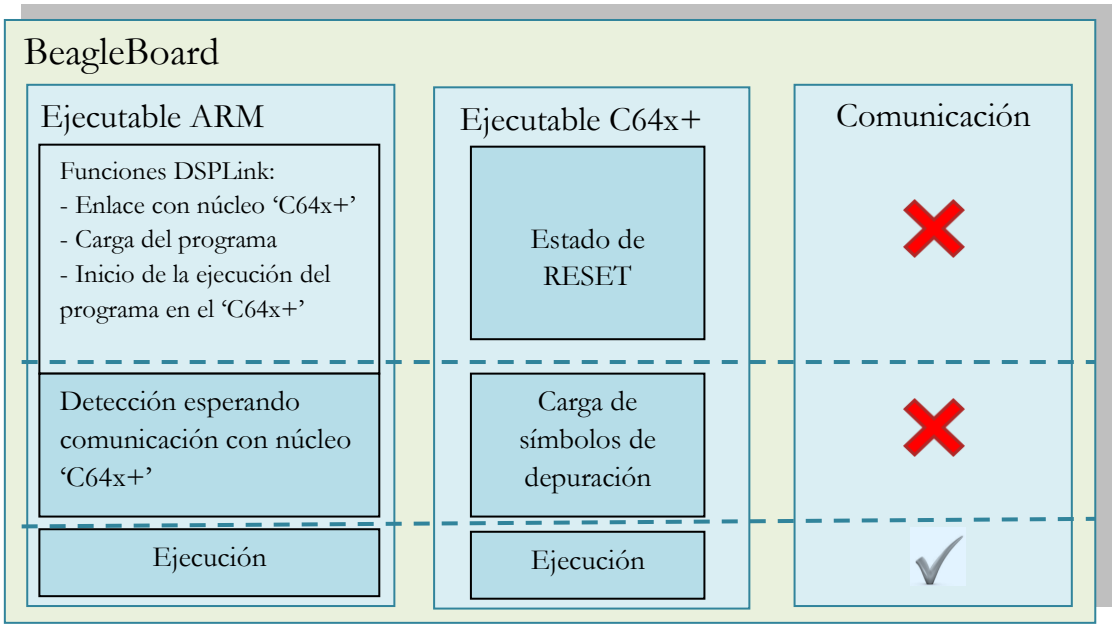

Fig. 40 Procedimiento de conexión entre los núcleos del procesador OMAP3530

En cuanto a particularidades observadas en el modo depuración, ha cobrado interés el tema de los diferentes tipos de puntos de ruptura que existen en CCS. El entorno de desarrollado integrado proporciona soporte a dos tipos de puntos de ruptura en función del tipo de depuración realizada: si se realiza una depuración *software* mediante la utilidad *gdb,* los puntos de ruptura se denominan *C/C++ brakpoints* mientras que si se decide hacer una depuración *hardware* los puntos de ruptura se definen con el nombre *CCS breakpoints*. Por tanto, se dan incompatibilidades cuando por ejemplo se coloca un punto de ruptura C/C++ en un proyecto CCS (caso del núcleo 'C64x+') que emplee depuración *hardware* pero por el contrario no se dan al colocar un punto de ruptura CCS en un proyecto C (caso del núcleo 'ARM') que realice el mismo tipo de depuración. Para sintetizar este tema, se ha confeccionado la siguiente tabla (ver Tabla 6) a modo aclaratorio:

Tipo de puntos de ruptura	Proyecto CCS	Proyecto C
***CCS breakpoints* (*Hardware*)**	Si	Si
***C/C++ breakpoints* (*Software*)**	No	Si

Tabla 6 Clasificación de puntos de ruptura en función del proyecto

Para habilitar/deshabilitar los puntos de ruptura y sus diferentes tipos se debe acudir a la pestaña *Run*, y dentro de ella a la categoría *Breakpoint Types* en la perspectiva de depuración de CCS.

Con este último detalle comentado, finaliza el estudio de las dos tareas básicas mencionadas al comienzo de este apartado acerca de la adaptación e importación de proyectos y su posterior depuración, ambas en CCS, por lo que únicamente faltan por ver un aspecto más en la configuración del sistema completo antes de entrar en el desarrollo principal de este Trabajo Fin de Máster: la elaboración de *scripts* de configuración para dotar al sistema completo de mayor agilidad y eficiencia.

3.4 *Scripts* de configuración

La elaboración de *scripts* permite automatizar procesos y ser más eficientes en el conjunto global de la aplicación desarrollada. Generalmente, las órdenes que se encuentran en un *script* son instrucciones que se repiten en numerosas ocasiones y es conveniente crear ficheros que únicamente requieran ser llamados desde una terminal de consola en vez de introducirlos una y otra vez por separado. Los *scripts* pueden ser identificados a través de:

- Encabezamiento: la primera línea que identifica a un *script* tiene el siguiente contenido: `#! /bin/bash` y se denomina *shebang*.
- Extensión del fichero: aunque es opcional, los *scripts* tienen la extensión **.sh* con el fin de ser reconocidos fácilmente sin necesidad de abrir el fichero.

En este Trabajo Fin de Máster la creación de *scripts* ha contrarrestado de alguna forma la lentitud en la de generación de ejecutables para los núcleos del procesador OMAP3530, permitiendo realizar operaciones en paralelo que acelerasen el desarrollo final. En concreto se han definido seis *scripts*, de los cuales cuatro se ejecutan en el *host* Linux y los dos restantes lo hacen desde el Linux embebido en el núcleo 'ARM'. Algunos de ellos tienen un uso menor ya que formaron parte del desarrollo inicial de este trabajo, pero otros se convirtieron en piezas imprescindibles dentro de la aplicación desarrollada. A continuación se presentarán los contenidos de dichos *scripts* (ver figuras de la Fig. 41 a la Fig. 46), destacando su nombre, su lugar de ejecución (*host* o núcleo 'ARM'), su función principal y por último su usabilidad representada a través de una batería con diferentes niveles: nivel bajo, para indicar que fue usado en las etapas iniciales de desarrollo, nivel medio, para indicar que ha tenido un uso moderado; y nivel alto para indicar que ha sido ampliamente usado en todas las fases del trabajo desarrollado.

Nombre *script*	Ejecución	Funcionalidad	Uso
/home/user/ /SCRIPTS/ dsplink_modulos.sh	*Host*	Copia de módulos al directorio del sistema de ficheros Angström donde el *kernel* Linux embebido acudirá para cargar dichos módulos automáticamente	

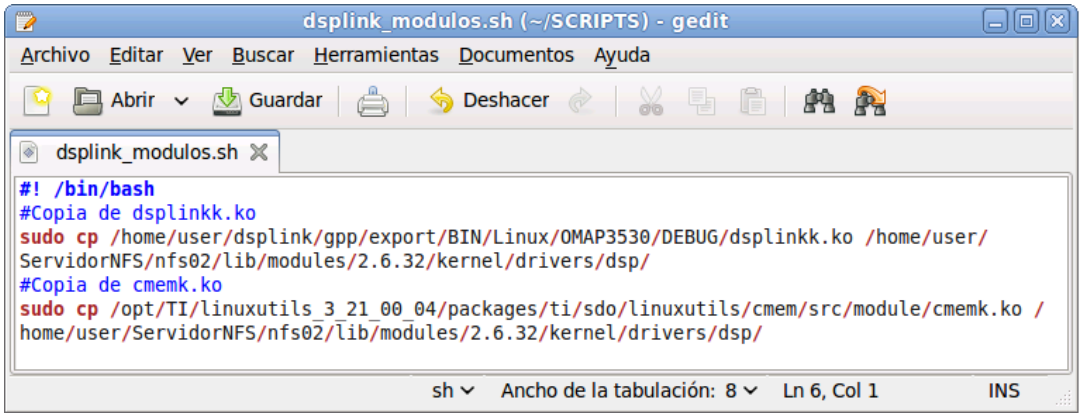

Fig. 41 *Script* de configuración *dsplink_modulos.sh*

Nombre *script*	Ejecución	Funcionalidad	Uso
/home/user/ ServidorNFS/nfs02/ usr/share/ti/gst/ omap3530/ load_modules.sh	'ARM'	Carga de módulos en el *kernel* Linux embebido realizada automáticamente en el arranque del sistema operativo del núcleo 'ARM'.	

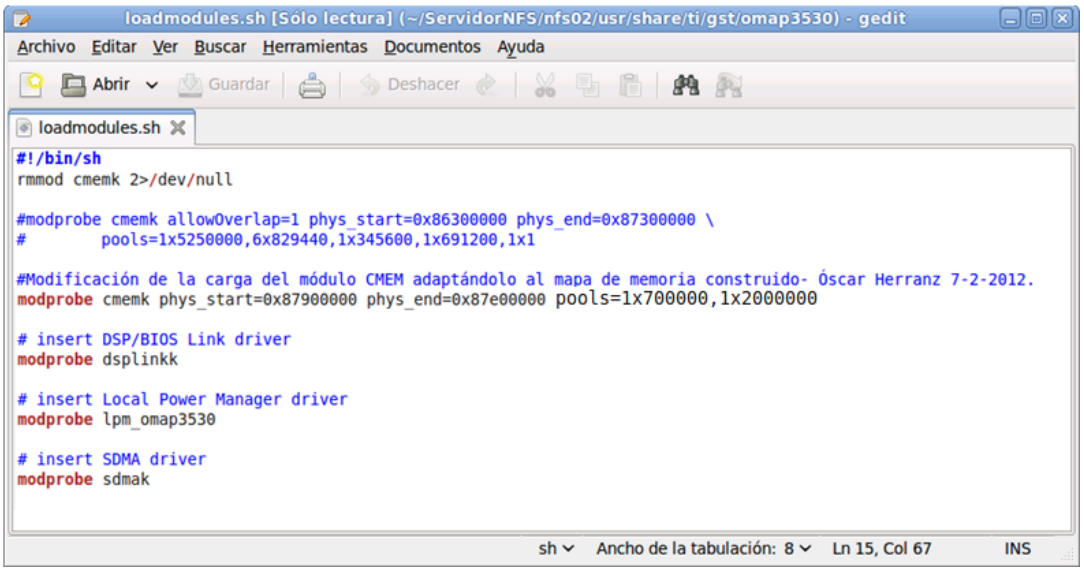

Fig. 42 *Script* de configuración *load_modules.sh*

Nombre *script*	Ejecución	Funcionalidad	Uso
/home/user/ /SCRIPTS/ copy_executables.sh	*Host*	Copia de los ejecutables de los núcleos 'ARM' y 'C64x+' al sistema de ficheros Angström para su posterior ejecución.	

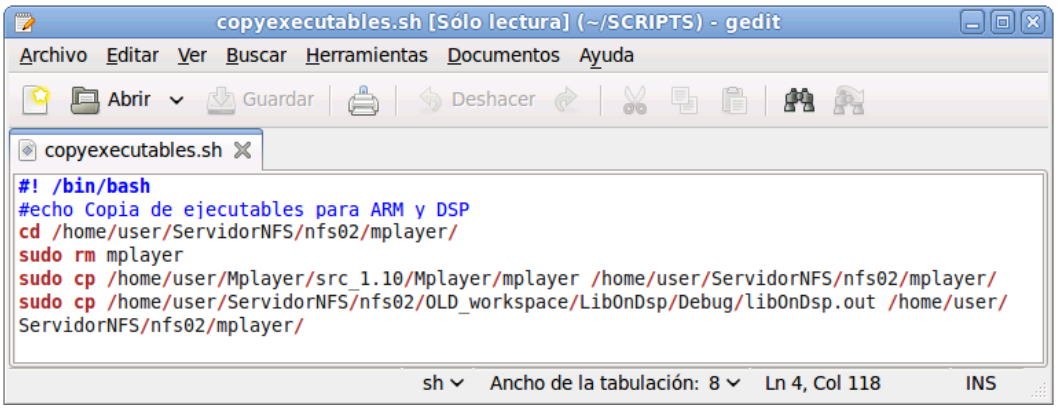

Fig. 43 *Script* de configuración *copyexecutables.sh*

Nombre *script*	Ejecución	Funcionalidad	Uso
/home/user/ ServidorNFS/nfs02/ scripts/ mplayer_execution.sh[21]	'ARM'	Arranque del servidor *gdb* en el núcleo 'ARM' para realizar la depuración de los dos núcleos del procesador OMAP3530 simultáneamente.	

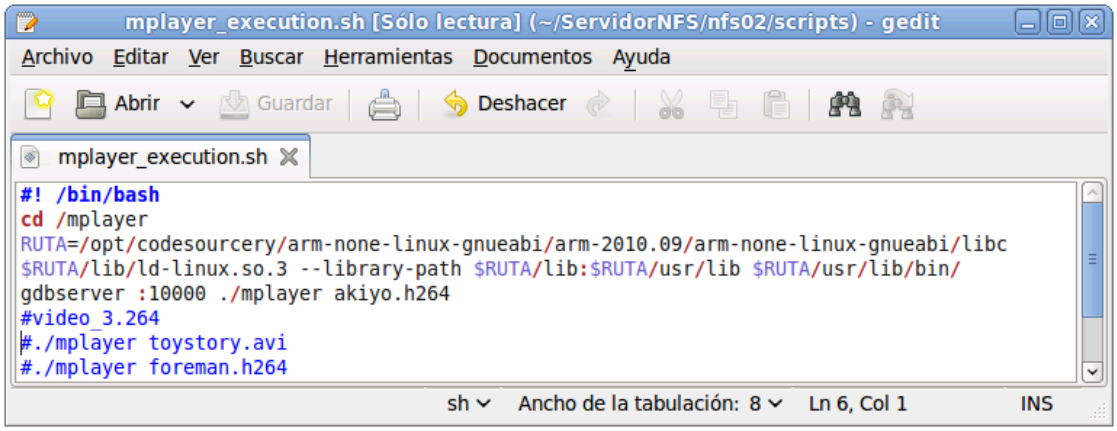

Fig. 44 *Script* de configuración *mplayer_execution.sh*

Nombre *script*	Ejecución	Funcionalidad	Uso
/home/user/ /SCRIPTS/ nfs_restart.sh	*Host*	Reiniciar el servidor NFS debido a problemas de accesibilidad al sistema de ficheros Angström a través de este protocolo.	

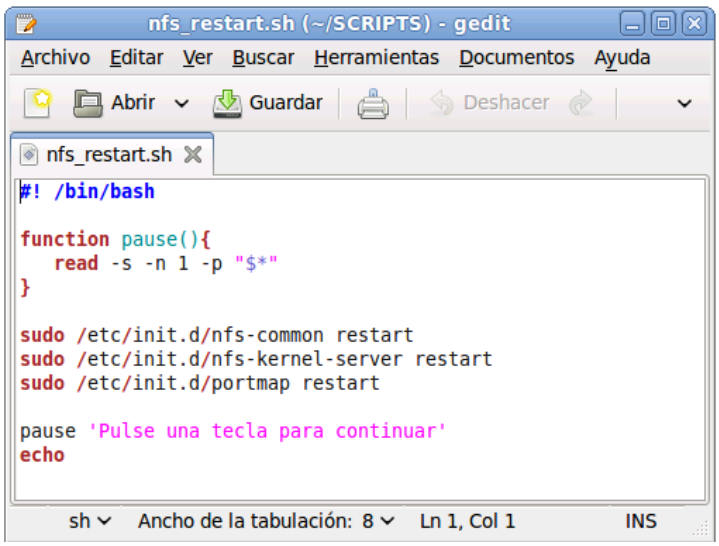

Fig. 45 *Script* de configuración *nfs_restart.sh*

[21] Ver apartado 3.3.2.1 para consultar el significado del contenido del script mplayer_execution.sh.

Nombre *script*	Ejecución	Funcionalidad	Uso
/home/user/ /SCRIPTS/ conf_mplayer.sh	*Host*	Configuración inicial del reproductor MPlayer para adaptarlo al entorno de trabajo empleado.	

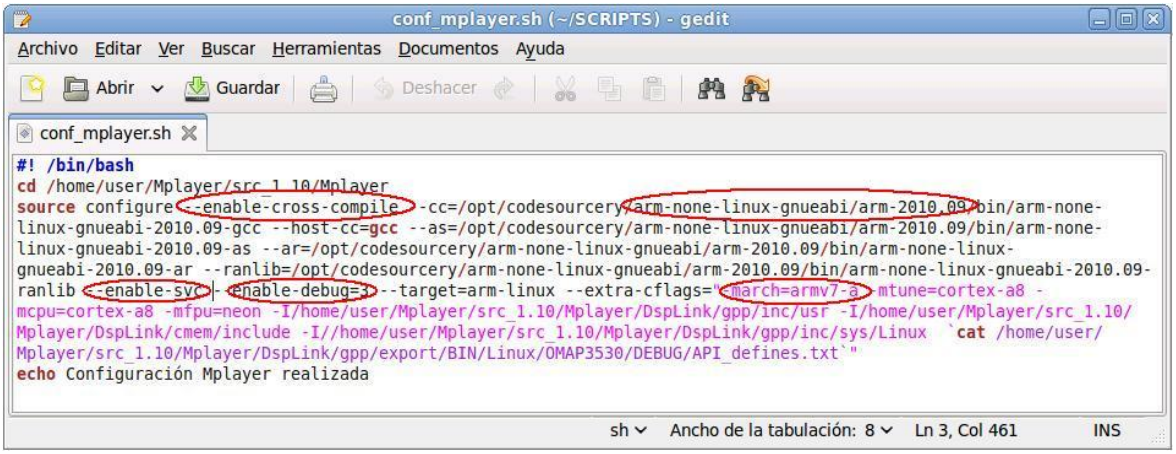

Fig. 46 *Script* de configuración *conf_mplayer.sh*

Dada la importancia del contenido de este último *script*, se han resaltado los campos más significativos que permitirán el empleo de la *toolchain* del fabricante CodeSourcery en un entorno de compilación cruzada; la habilitación del descodificador OpenSVC integrado en MPlayer; la habilitación de opciones de depuración en el ejecutable generado y por último, la arquitectura del procesador al que va dirigido. Para más información acerca de la configuración inicial del reproductor multimedia acudir a [1].

4 DESARROLLO

En este capítulo se presenta el análisis y las modificaciones realizadas en las diferentes etapas de la aplicación desarrollada en este Trabajo Fin de Máster.

En primer lugar se realiza una breve introducción acerca de la estructura a seguir a lo largo del capítulo, dividiendo el ciclo de desarrollo en las fases previa al descodificador (etapa de entrada de MPlayer), descodificación y posterior al descodificador (etapa de salida de MPlayer).

Para la fase previa al descodificador, se estudian procesos iniciales como la identificación del flujo de datos del fichero a descodificar así como la adaptación de datos realizada por el demultiplexor, actuando como interfaz de entrada al descodificador.

Para la fase propia de descodificación se analizan los cuatro elementos principales que intervienen en ella: DSPLink, CMEM, MPlayer y OpenSVC, destacando de los dos primeros sus respectivas API, mientras que de los dos últimos se detallan las modificaciones que ha sido necesario realizar para la integración. La descripción de la integración de MPlayer y OpenSVC con la acción de los módulos Linux finaliza esta fase del desarrollo.

Para la fase posterior al descodificador se distinguen los procesos involucrados en la representación en un monitor de las imágenes descodificadas. De este modo, se analiza en primer lugar la configuración del controlador de salida de vídeo, posteriormente se estudia la adaptación de la información de la imagen descodificada en el núcleo 'ARM' y para terminar se presentan las funciones encargadas de la presentación.

4.1 Introducción

Presentadas las diferentes tecnologías tanto *software* como *hardware* y analizadas en profundidad las configuraciones de cada una de ellas, es el momento idóneo para abordar el desarrollo de este Trabajo Fin de Máster: la integración del reproductor multimedia MPlayer en el núcleo 'ARM' del procesador OMAP3530 con el descodificador OpenSVC alojado en el núcleo 'C64x+' del mismo. En primer lugar, es importante destacar que el núcleo principal de este trabajo gira en torno al descodificador, por lo que las modificaciones realizadas en su estructura, la comunicación e intercambio de información entre núcleos, etc. supondrán buena parte del desarrollo de este capítulo. Sin embargo, esa integración mencionada anteriormente también requiere de un análisis detallado tanto de la etapa de entrada al descodificador como de la etapa de salida de éste. Ambas etapas se llevan a cabo en el reproductor MPlayer, por lo que cómo se suministra la información al descodificador y cómo se obtiene la información de éste para su posterior presentación se convierten en procesos de significativa importancia. En consecuencia, se van a describir en profundidad las tres etapas, dedicando un mayor estudio y análisis a la segunda de ellas, la integración del descodificador OpenSVC en el procesador digital de señal del procesador OMAP3530.

Dada la importancia de los temas a tratar y la extensión del capítulo, se ofrece a continuación un resumen acerca de la organización de éste, destacando las tareas más importantes que se realizan en cada fase. Antes de entrar en el detalle de las fases iniciales, se presenta un diagrama general de la aplicación desarrollada (ver Fig. 47), con el fin de situar de forma gráfica los diferentes elementos que la forman pese a que no se conozca cómo están implementados por ahora. Además, esa figura contiene una serie de números que refleja el orden secuencial de las tareas realizadas.

A continuación, el capítulo se ha dividido en tres fases:

- Etapa de entrada del descodificador: en este fase se estudia cómo MPlayer administra y adapta la información del fichero a procesar para que el descodificador la reciba correctamente.
- Descodificación: en este fase se presentan las modificaciones realizadas en los cuatro elementos principales de este trabajo: MPlayer, OpenSVC, DSPLink y CMEM. Analizados esos cambios, se lleva a cabo la integración del reproductor con el descodificador, detallando el orden en el que se realizan las operaciones, aspecto muy importante en la realización de este Trabajo Fin de Máster.
- Etapa de salida del descodificador: en esta fase nuevamente se estudia el método de trabajo que implementa MPlayer para recoger la información de la imagen descodificada y representarla en un monitor a través del controlador de salida de vídeo *Framebuffer*. Asociado a esa representación, también se describe la conversión *software* realizada entre espacios de color para una correcta visualización de la imagen.

En las tres fases se ha incluido información perteneciente al código de las funciones que implementan las diferentes fases descritas, con comentarios adicionales que faciliten la

comprensión de los contenidos. Además, también se han realizado diagramas de flujo para visualizar el orden de las tareas en procesos complejos como la integración o funciones que entrañen dificultad.

Destacar que la filosofía de trabajo siempre ha sido la de adaptar la información a la metodología de trabajo de MPlayer, de ahí que en la primera y tercera fase se haya realizado únicamente un análisis en profundidad del funcionamiento del reproductor, mientras que en la segunda fase ha sido necesario estudiar y modificar el método de trabajo del descodificador OpenSVC alojado en el núcleo 'C64x+' para realizar exitosamente la integración.

Definidos los contenidos del presente capítulo, se hace referencia nuevamente al diagrama general (ver Fig. 47), donde los números representan el orden secuencial de las operaciones a realizar para comunicar y transferir información entre los núcleos del procesador OMAP3530 correctamente.

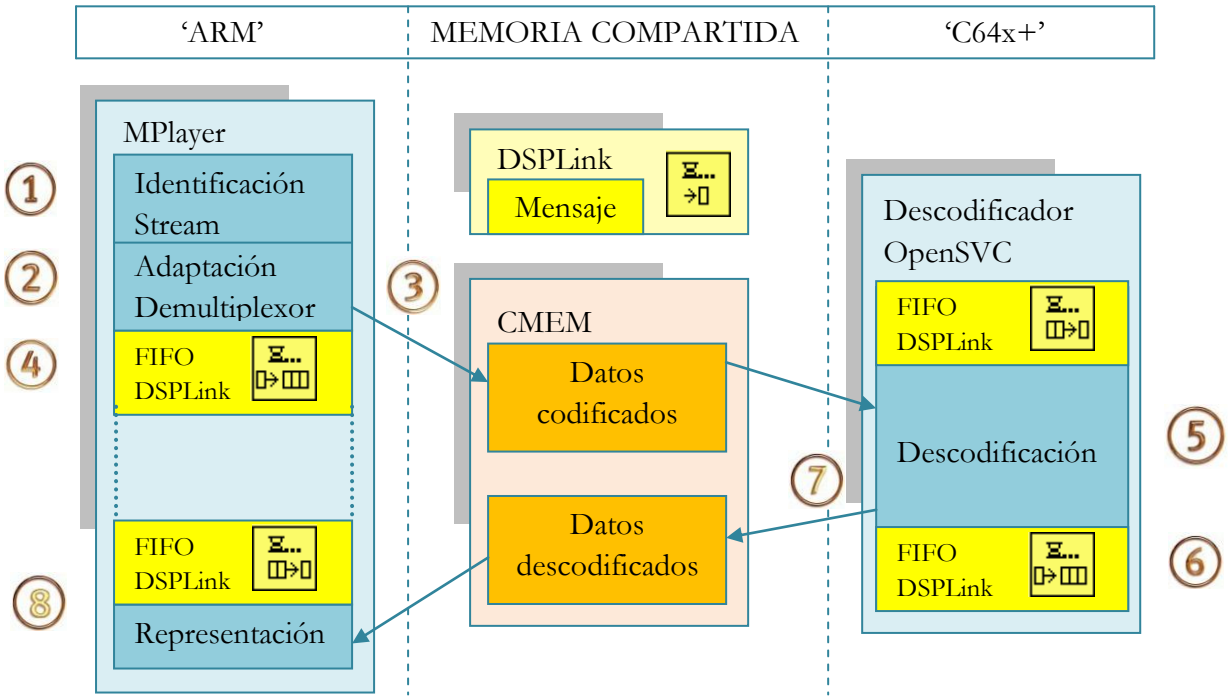

Fig. 47 Diagrama general de la aplicación desarrollada

En concreto, esas operaciones y su orden se definen a continuación[22]:

1. Identificación de los diferentes *streams* del fichero a descodificar ('ARM').
2. Adaptación de la información por parte del demultiplexor ('ARM').
3. Copia en CMEM (1ª *pool*) de un fragmento de información perteneciente a los datos codificados del fichero.
4. Envío de mensaje a través de la cola de mensajes FIFO (*First In First Out*) implementada por DSPLink ('ARM').
5. Recepción del mensaje y procesamiento de la información codificada por OpenSVC ('C64x+').

[22] Entre paréntesis se indica el núcleo del procesador OMAP3530 donde se lleva a cabo la operación ('ARM' o 'C64x+').

6. Copia en CMEM (2ª *pool*) de la imagen descodificada por OpenSVC en caso de haberla ('C64x+').
7. Envío de mensaje a través de la cola de mensajes FIFO implementada por DSPLink ('C64x+').
8. Recepción del mensaje y presentación en un monitor tras previa adaptación de la información de la imagen y del controlador de salida de vídeo ('ARM').

Además de las tres etapas comentadas anteriormente que hacen referencia a la entrada del descodificador (números 1 y 2), su salida (número 8) y el proceso de descodificación (número 5), la Fig. 47 también muestra el empleo de la cola de mensajes FIFO (First In First Out) a través de DSPLink (números 4 y 7) y el uso de la zona de memoria conocida por ambos núcleos a través de CMEM (números 3 y 6). Más adelante, se detallarán cómo se han realizado procesos imprescindibles como la comunicación, la sincronización o el intercambio de datos entre los núcleos del procesador OMAP3530. Para ello, esta figura se irá empleando a lo largo del capítulo con el fin de no perder en ningún momento cuál es la estructura de la aplicación construida. También se proporcionará información de más bajo nivel, perteneciente a diferentes secciones de código de la aplicación diseñada, para que este Trabajo Fin de Máster pueda servir de referencia a futuros trabajos del Grupo de Investigación GDEM.

4.2 Etapa de entrada del descodificador

El análisis acerca de cómo MPlayer administra y adapta la información para que el descodificador pueda realizar su trabajo es complejo e involucra a muchos ficheros y estructuras dentro del propio código de la aplicación. Al tratarse de una tarea sin desarrollo previo dentro del Grupo de Investigación GDEM, ha supuesto un gran esfuerzo comprender cómo se gestiona esa etapa de entrada, de modo que se procederá a continuación a su análisis apoyándose en diagramas de flujo y secciones de código comentadas. En primer lugar, la siguiente figura (ver Fig. 48) muestra los diferentes bloques interconectados por los que está compuesta esta etapa de entrada:

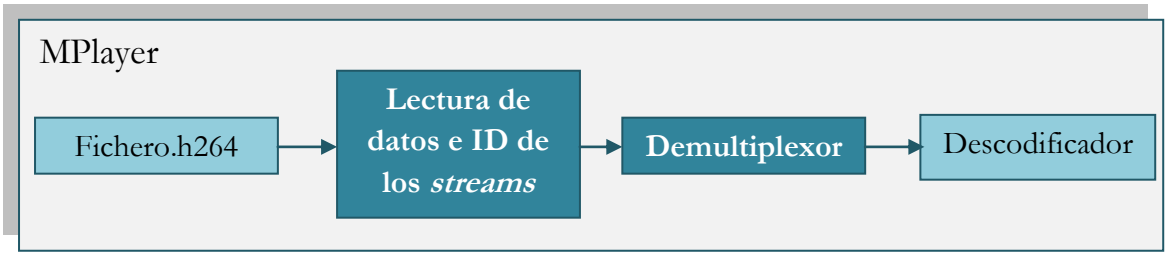

Fig. 48 Esquema de la etapa de entrada del descodificador

Se ha optado por dividir esta etapa anterior a la descodificación principalmente en dos sub-etapas:

- Identificación del *stream*, donde se describirá cómo MPlayer es capaz de reconocer los diferentes *streams* que contiene el fichero a descodificar a través de la lectura de bloques de datos de tamaño constante.

- Adaptación del demultiplexor, donde se explicará la adaptación realizada desde que MPlayer ha reconocido los diferentes *streams* hasta que esa información es enviada al descodificador para su procesamiento en bloques de datos de tamaño variable.

Además, MPlayer proporciona una función dentro de su estructura: `void mp_msg(int, int, char*)` ubicada en `/MPlayer/mp_msg.c` que realizando una simple modificación permite habilitar la impresión de mensajes por la terminal de consola donde se ejecute la aplicación. Esta utilidad ha sido de gran ayuda para la depuración del reproductor, analizando los diferentes mensajes obtenidos.

La Fig. 49 muestra la parte de la función dónde se debe habilitar/deshabilitar dicha utilidad

```
void mp_msg(int mod, int lev, const char *format, ... ){

    if (!mp_msg_test(mod, lev)) return; //Comentar si se desean habilitar los mensajes MPlayer
    va_start(va, format);
    vsnprintf(tmp, MSGSIZE_MAX, format, va);
```

Fig. 49 Sección de código que habilita la impresión mensajes MPlayer (función mp_msg)

Haciendo referencia al diagrama inicial del capítulo, el análisis que se va a realizar a continuación se muestra resaltado (color azul) respecto al resto de la figura (ver Fig. 47).

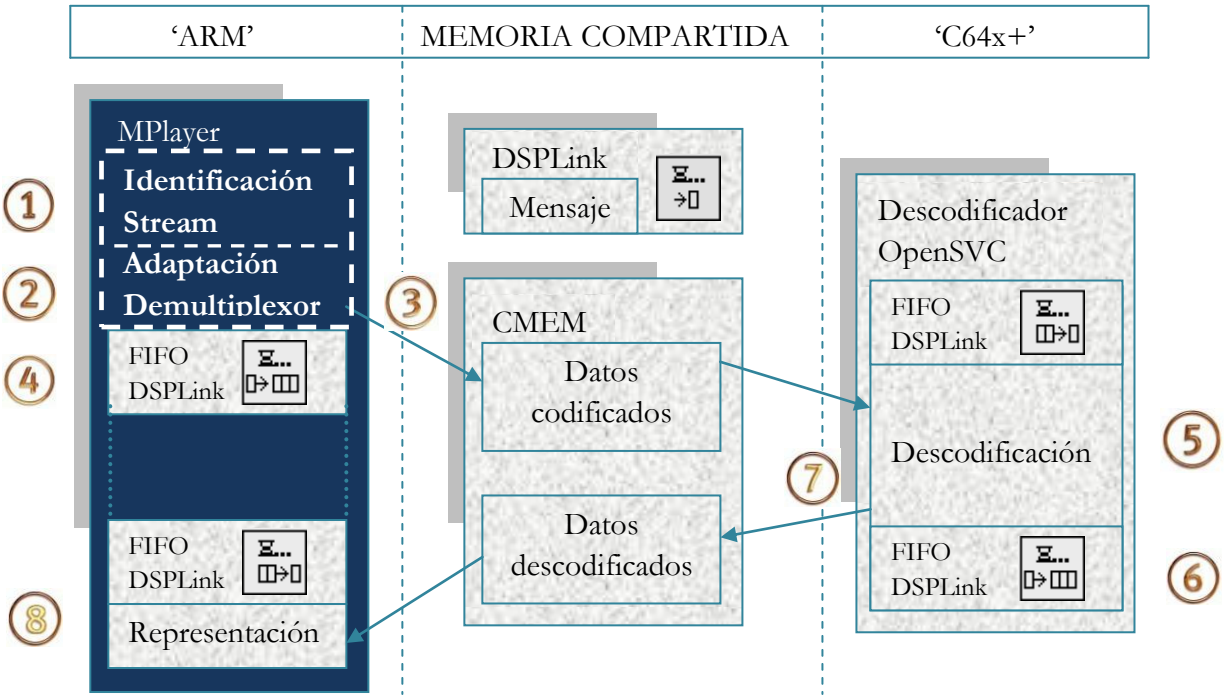

Fig. 47 Diagrama general de la aplicación desarrollada

La funcionalidad de cada sub-etapa está claramente diferenciada, pero los procesos que se realizan internamente en cada una de ellas sí que se encuentran relacionados. Se comenzará por describir cómo se identifica el/los *stream*/s y después se explicará el proceso de adaptación del demultiplexor.

4.2.1 Identificación del *stream*

MPlayer proporciona soporte para diferentes medios de entrada: fichero, dvd, *streaming* por red, etc. y en función del tipo de entrada, realiza su lectura por bloques de tamaño variable, por sectores, o una cantidad determinada de *bytes* entre otros modos. En el caso de este Trabajo Fin de Máster, como se citó en la introducción del reproductor (apartado 2.5), se va a trabajar con ficheros con extensión **.h264* y codificados por el estándar de codificación escalable H.264/SVC. Una vez que MPlayer identifica el tipo de entrada, la cual es introducida a través de una línea de comandos (ver apartado 3.4, *mplayer_execution.sh*), el reproductor comienza el proceso de reconocimiento de *streams*.

En este punto es importante destacar que los ficheros empleados en este trabajo incluyen exclusivamente la información del codificador, a diferencia de lo que sucede en la mayoría de aplicaciones multimedia clásicas, donde los flujos de vídeo y audio se comprimen de forma independiente formando lo que se denomina *Elementary Stream* (ES). A continuación, estas unidades ES son multiplexadas en algún formato de encapsulado, de manera que en el proceso inverso de demultiplexación se reconozca rápidamente el tipo de fichero, y una vez identificado éste, se lleve a cabo la separación de los *streams* de audio y vídeo contenidos en el fichero.

Para la identificación del tipo de fichero, MPlayer contiene una serie de analizadores (*parsers*) que se encargan de realizar esta función[23]. Sin embargo, como se dijo anteriormente, los ficheros empleados en este trabajo (**.h264*) no presentan ningún formato de encapsulado que permita obtener esa información, por lo que MPlayer implementa otro método para identificar el formato del fichero que se verá a continuación.

Recordando los tipos de lectura que realiza MPlayer en función del tipo de entrada, para operaciones con ficheros se realiza una lectura de una cantidad fija de *bytes*, en concreto de bloques de 2 KB[24]. Esta operación se realiza en la función `int fill_buffer (stream_t *, char *, int)` ubicada en el fichero `/MPlayer/stream/stream_file.c`. Destacar que el segundo parámetro de la función es un puntero de tipo *char*, esto es, la dirección de memoria a partir de la cual se comenzará a escribirse el bloque de 2 KB demandado.

La Fig. 50 muestra el contenido de la función descrita recientemente.

```
static int fill_buffer(stream_t *s, char* buffer, int max_len){
  int r = read(s->fd,buffer,max_len);   //fd: manejador del fichero de entrada
                                        //buffer: dirección de memoria destino
                                        //max_len: bytes a copiar, 2048
  return (r <= 0) ? -1 : r;
}
```

Fig. 50 Sección de código que copia bloques de tamaño 2048 B desde el fichero a descodificar (función fill_buffer)

Aunque las lecturas de fichero se realicen en bloques de 2 KB, MPlayer integra esta función en un proceso iterativo de mediana complejidad para copiar un mayor volúmen de información y mejorar la eficiencia del proceso. En concreto, tanto la obtención del

[23] En el fichero formats.txt (ubicado en /Mplayer/DOCS/tech/) se pueden encontrar los diferentes analizadores de *streams* que soporta el reproductor multimedia.
[24] 2 KB = 2 KBytes = 2048 bytes.

puntero (donde se copiarán los datos) mencionado como el tamaño de datos a copiar se generan en la función `void fill_buffer (ByteIOContext *)` ubicada en el fichero dentro de la estructura del reproductor `/MPlayer/libavformat/aviobuf.c`.

La Fig. 51 muestra la sección de código donde se obtienen ambos parámetros.

```
static void fill_buffer(ByteIOContext *s)
{
    uint8_t *dst= !s->max_packet_size && s->buf_end - s->buffer < s->buffer_size ? s->buf_ptr : s->buffer;
    int len= s->buffer_size - (dst - s->buffer);
    int max_buffer_size = s->max_packet_size ? s->max_packet_size : IO_BUFFER_SIZE;
```

Fig. 51 Obtención del tamaño y dirección de memoria empleados en el proceso iterativo (función fill_buffer)

El tamaño de datos a copiar se va incrementando a través del código de esta función y asciende a 32 KB[25]. Esto no significa que se copie directamente esta cantidad de datos desde fichero sino que se alcanzará esa cifra mediante sucesivas copias de 2 KB[26]. Para llevar a cabo esta operación, es fácil pensar en la implementación de un proceso iterativo que se ejecute hasta alcanzar la cantidad de datos requerida, en este caso 32 KB. MPlayer realiza esta tarea en la función `int stream_read (stream_t *, char *, int)` ubicada en el fichero `/MPlayer/stream/stream.h`. En esta función, exceptuando el primero, los otros dos parámetros de entrada se corresponden con los generados en la función `fill_buffer` (ver Fig. 51).

A continuación se analizará el contenido de esta función mediante la presentación de su código con comentarios adicionales (ver Fig. 52).

```
inline static int stream_read(stream_t *s,char* mem,int total){
  int len=total;                //Copia local del tamaño a copiar: 32768 KB
  while(len>0){                 //Proceso iterativo ejecutado hasta alcanzar la cantidad demandada
    int x;
    x=s->buf_len-s->buf_pos;    //Actualizacion del índice que determina el estado del buffer
                                //buf_len representa la longitud del buffer: 2048
                                //buf_pos representa la posición actual del buffer
    if(x==0){                   //Si se han procesado los datos, se obtiene un nuevo bloque de datos...
                                //...mediante la función fill_buffer (fichero stream_file.c)
      if(!cache_stream_fill_buffer(s)) return total-len; // EOF
      x=s->buf_len-s->buf_pos;  //Actualización del índice. Datos leídos: 2048 B
    }
    if(s->buf_pos>s->buf_len) mp_msg(MSGT_DEMUX, MSGL_WARN, "stream_read: WARNING! s->buf_pos>s->buf_len\n");
    if(x>len) x=len;            //Ajuste para la última iteración del bucle
    memcpy(mem,&s->buffer[s->buf_pos],x);   //Copia de 2048 B a partir de la dirección de memoria generada...
                                            //...en la función fill_buffer (fichero aviobuf.c)
    s->buf_pos+=x; mem+=x; len-=x;          //Actualización de variables para la siguiente iteración
  }
  return total;                 //Se devuelven los datos procesados totales
}
```

Fig. 52 Sección de código que copia bloques de tamaño 32768 B desde el fichero a descodificar (función stream_read)

En ella se representa el proceso iterativo mencionado, el cual finalizará cuando se haya copiado el total de datos ordenado por el parámetro de entrada *total*. La actualización de las variables tras cada iteración es clave para la correcta copia de información.

[25] 32 KB = 32 KBytes = 32768 bytes.

[26] Esa cantidad de datos, 32 KB, es constante salvo en el caso descrito en páginas posteriores dentro del apartado 4.2.1.

Hasta aquí, el proceso descrito en su totalidad únicamente adquiere una cantidad de datos fija del fichero a descodificar y los ubica en un posición de memoria conocida, tarea insuficiente para reconocer los diferentes *streams* que contiene el fichero. En consecuencia, es necesario un procesamiento mayor para llevar a cabo esa tarea, con algo más "de inteligencia" que la simple copia de un bloque constante de datos. En este punto, aparece el significativo concepto de paquete (*packet*) con el que MPlayer trabajará en las fases posteriores y que será el utilizado para identificar los *streams* del fichero.

Por tanto, parece obvio preguntarse qué se hace con la información adquirida de fichero. Pues bien, una vez copiado un bloque de 32 KB en memoria, se realiza un procesamiento de esa información de tal forma que a partir de ella se obtienen paquetes de tamaño variable, es decir, conjuntos de datos pero ahora de longitud variable a diferencia de lo que sucedía antes. Al no disponer el fichero de entrada de ningún formato de encapsulado, MPlayer necesita procesar un determinado número de paquetes para identificar el formato del vídeo codificado. En cuanto a la longitud variable de los paquetes, en un principio se desconocía el por qué de esa longitud, la cual puede ser superior o inferior al tamaño de datos adquirido de fichero, 32 KB, pero cuando se analizó el proceso de descodificación se observó que la longitud de los paquetes guarda una estrecha relación con las NALs encontradas en cada paquete, en concreto se comprobó que cada paquete contenía un número exacto de unidades NAL.

En consecuencia con lo anterior, ha sido necesario analizar cómo se lleva a cabo esa creación de paquetes no solo para este proceso inicial de identificación de *streams* sino también como parte del bucle general de funcionamiento de MPlayer. Por esta razón, a continuación se describirá desde cómo se genera un paquete y las diferentes operaciones que se realizan con él hasta que se da por terminada esta etapa de reconocimiento.

Nuevamente, el proceso a nivel de funciones y estructuras que forman parte del mismo es elevado, de modo que se tratará de simplificar para tener una idea clara del funcionamiento general ya que éste no es uno de los objetivos de este Trabajo Fin de Máster. Por ello, la función que se va a tomar como punto de partida ya recibe los parámetros necesarios para realizar las tareas asociadas a paquetes. Esta función se denomina int `av_get_packet(ByteIOContext *, AVPacket, int)` y se ubica en el siguiente fichero de la estructura del reproductor `/MPlayer/libavformat/utils.c`. La Fig. 53 muestra el contenido de la función empleada como punto de partida.

```
int av_get_packet(ByteIOContext *s,AVPacket *pkt, int size)
{
    int ret= av_new_packet(pkt, size);      //Obtención del puntero al campo de datos del paquete
    if(ret<0)
        return ret;

    pkt->pos= url_ftell(s);

    ret= get_buffer(s, pkt->data, size);    //Copia en memoria y por consiguiente creación del paquete...
                                            //...en la dirección apuntada por la función av_new_packet...
                                            //y tamaño obtenido a través de la variable size
    if(ret<=0)
        av_free_packet(pkt);
    else
        av_shrink_packet(pkt, ret);

    return ret;
}
```

Fig. 53 Sección de código que lanza el proceso de generación de paquete (función av_get_packet)

En ella puede verse como la función ya recibe como parámetros entre otros un puntero a la estructura del paquete, cuyos campos principales son los datos y el tamaño de éstos; y también recibe el tamaño del paquete que será empleado en las funciones resaltadas sobre la figura. En cuanto a éstas, la primera de ellas, `int av_new_packet (AVPacket *, int)` ubicada en el fichero `/MPlayer/libavcodec/avpacket.c`, realiza operaciones de inicialización y adaptación de los diferentes campos del paquete para que sean usados en funciones posteriores, mientras que la segunda realiza la copia de paquetes empleando los parámetros de la primera.

La Fig. 54 presenta el contenido de la función `av_new_packet` con información adicional.

```
int av_new_packet(AVPacket *pkt, int size)
{
    uint8_t *data= NULL;
    if((unsigned)size < (unsigned)size + FF_INPUT_BUFFER_PADDING_SIZE)
        data= av_malloc(size + FF_INPUT_BUFFER_PADDING_SIZE);      //Obtención del puntero
    if (data){
        memset(data + size, 0, FF_INPUT_BUFFER_PADDING_SIZE);
    }else
        size=0;

    av_init_packet(pkt);          //Inicialización de la estructura del paquete
    pkt->data = data;             //Copia del puntero obtenido al campo data de la estructura del paquete
    pkt->size = size;             //Copia del tamaño recibido como parámetro de entrada al campo size de la...
                                  //...estructura del paquete
    pkt->destruct = av_destruct_packet;
    if(!data)
        return AVERROR(ENOMEM);
    return 0;
}
```

Fig. 54 Sección de código que realiza la inicialización y adaptación de los campos del paquete (función av_new_packet)

Respecto a la función `int get_buffer (ByteIOContext *, char *, int)`, ésta se ubica en el siguiente fichero `/MPlayer/libavformat/aviobuf.c` y tiene como objetivo copiar una determinada cantidad de datos (impuesta por la variable *size*) en la dirección de memoria obtenida en la función `av_new_packet`. Al tratarse de una función compleja con algunas particularidades se ha preferido realizar un diagrama de flujo (ver Fig. 55) en vez de analizar su código, ya que puede inducir a equivocaciones o malas interpretaciones fácilmente.

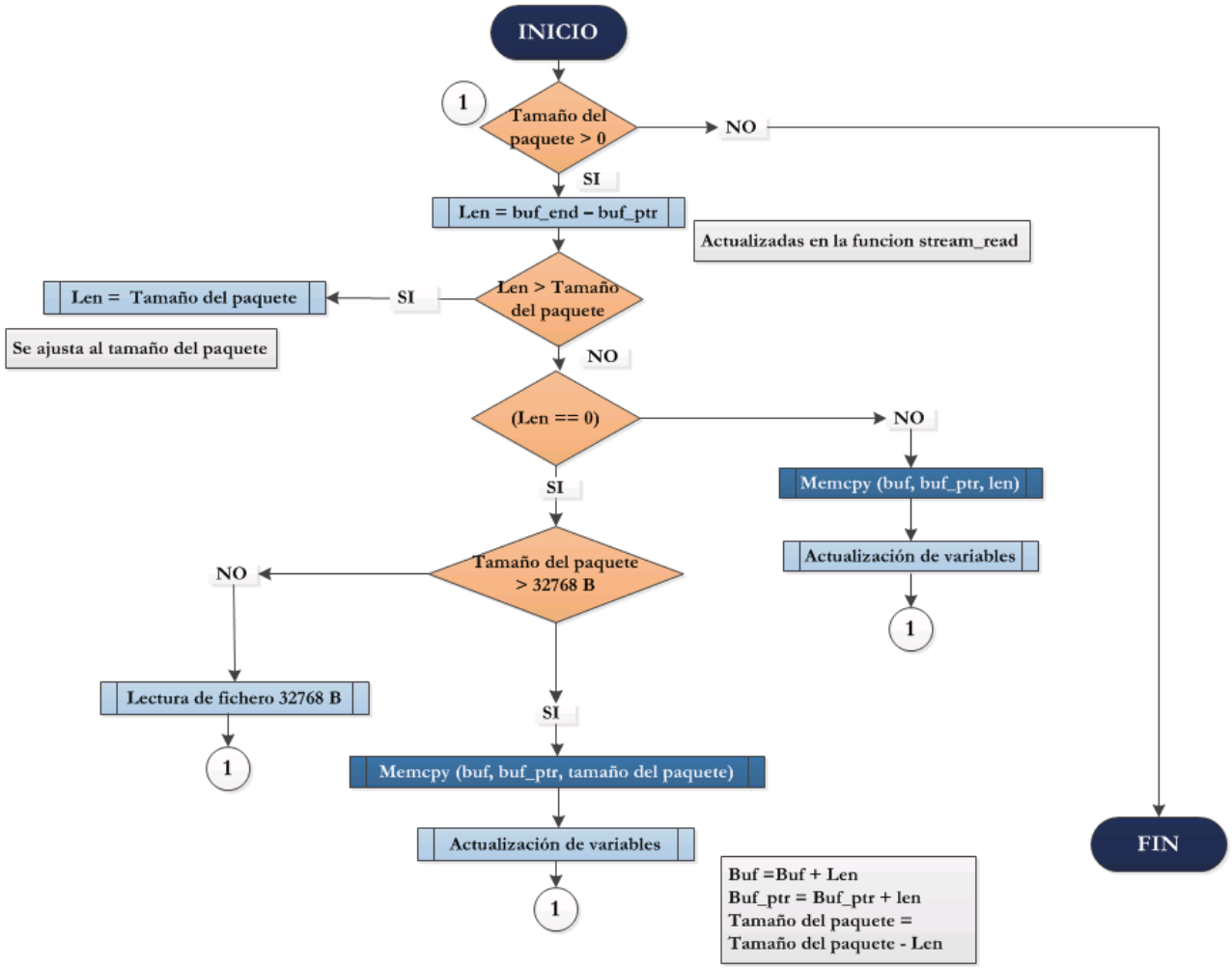

Fig. 55 Diagrama de flujo perteneciente a la función int get_buffer

El diagrama presentado muestra como se realiza la copia del paquete de dos maneras diferentes en función del tamaño del paquete. Nuevamente, se vuelve a tener un proceso iterativo que permanecerá en ejecución hasta que se haya copiado la longitud predefinidad del paquete. Además, se trabaja con una variable denominada *len* que guarda la diferencia entre los dos punteros empleados en la copia de datos desde fichero. El primero de ellos, denominado *buf_ptr,* apuntará a la dirección inicial de la fuente de datos y se irá incrementando tantas posiciones como número de *bytes* sean copiados; y un segundo puntero denominado *buf_end*, cuya dirección es la del primer puntero más la cantidad de datos copiados de fichero, es decir, 32 KB, y que no se incrementará hasta realizar una nueva copia de datos desde fichero (función `stream_read`).

A continuación se compara el valor de *len* obtenido con el tamaño del paquete actual y únicamente se realizan operaciones sí el tamaño del paquete es menor que la diferencia obtenida a través de los punteros, para lo cual se ajusta el campo *len* al tamaño del paquete. Una vez realizada esa operación, se comprueba el valor del campo *len,* obteniendo los siguientes resultados:

- Si vale cero, se contemplan dos posibles situaciones:
  - El tamaño del paquete es superior a 32768 B, de modo que se copia directamente la información de fichero (función `stream_read`) a la dirección de memoria apuntada por el puntero tantos *bytes* como tenga el tamaño del paquete.
  - Se realiza una copia de un bloque de 32768 B del fichero de entrada como la descrita en este apartado, actualizándose el valor de los campos *buf_ptr y buf_end* y se vuelve al bucle.

- Si es superior a cero se copia la información en el puntero que se pasa como parámetro a la función, realizándose dicha copia en la propia función `get_buffer`. El número de *bytes* copiados es igual al tamaño del paquete (el campo *len* se ajustó previamente al tamaño del paquete) y la fuente origen de datos se corresponde con la información almacenada en el proceso de copia de datos de fichero.

Destacar que en el otro proceso de copia comentado anteriormente, no se almacena ninguna información previa, sino que se escribe directamente la información desde el propio fichero tantos *bytes* como tenga el tamaño del paquete, siendo éste superior siempre a 32768 B para estas situaciones concretas.

Por último, se lleva a cabo la actualización de variables que permiten, cuando corresponda, romper el bucle gracias a las instrucciones *break* incluidas dentro del proceso iterativo.

Algunos ejemplos numéricos que dan lugar a diferentes situaciones se presenta en la siguiente tabla (ver Tabla 7) a modo aclaratorio.

Escenario	Valor del campo *len*	Tamaño del paquete	Comportamiento
1	500	300 B	Campo *len* se ajusta al tamaño del paquete y se copian los datos tras previa copia desde fichero.
2	0	800 B	Relleno de datos desde fichero y nueva iteración del proceso.
3	0	40000 B	Campo *len* no se ajusta al tamaño del paquete y se copian los datos directamente desde fichero.
4	500	800 B	No se ha contemplado con las secuencias .h264 probadas hasta la fecha

Tabla 7 Ejemplos de comportamiento de la función int get_buffer

Este proceso de inserción de la información en paquetes se repite hasta que MPlayer tiene la suficiente información como para identificar los diferentes *streams* que contiene el fichero. En este Trabajo Fin de Máster no se ha ido más allá para determinar cuál es la cantidad necesaria ya que no es el objetivo, pero sí que se ha detectado ese momento en el cual el reproductor está listo para operaciones posteriores.

La Fig. 56 muestra la sección correspondiente de código donde se produce dicha acción, en concreto la función `int av_find_stream_info(AVFormatContext*)`.

```
int av_find_stream_info(AVFormatContext *ic)
{
    if (!(ic->ctx_flags & AVFMTCTX_NOHEADER)) {
        /* if we found the info for all the codecs, we can stop */      //Información del reproductor MPlayer
        ret = count;
        av_log(ic, AV_LOG_DEBUG, "All info found\n");
        break;
    }
```

Fig. 56 Sección de código que controla el número de paquetes en el proceso de reconocimiento de *streams* (función av_find_stream_info)

La ejecución de esa sentencia permite al reproductor identificar los *streams*. Para este trabajo siempre se han empleado secuencias de vídeo escalable que únicamente contienen un *stream* de vídeo y ninguno de audio. Si la opción que proporciona MPlayer de mostrar mensajes está habilitada, se obtiene la siguiente información por la terminal de consola (ver Fig. 57).

```
==> Found video stream: 0
ID_VIDEO_ID=0
aspect= 352*0/(288*1)
[lavf] stream 0: video (h264), -vid 0
LAVF: 0 audio and 1 video streams found
LAVF: build 3425280
VIDEO:  [H264]  352x288  12bpp  25.000 fps    0.000 kbps
[V] filefmt:44  fourcc:0x34363248  size:352x288  fps:25.000  ftime:=0.0400
```

Fig. 57 Reconocimiento de *streams* por parte de MPlayer

A través de la terminal puede comprobarse como se ha materializado la identificación del *stream* de vídeo en este caso particular, de modo que MPlayer ya está preparado para la siguiente sub-etapa, denominada en el inicio de este apartado como adaptación del demultiplexor.

4.2.2 Adaptación del demultiplexor

El proceso que se va a describir a continuación tiene como objetivo separar audio y vídeo para que el demultplexor proporcione en la forma adecuada esos paquetes creados con anterioridad al descodificador, con el fin de que éste comience a realizar su tarea. En primer lugar, es importante destacar que cada uno de los paquetes obtenidos es almacenado en una lista de paquetes a través de la función `AVPacket *add_to_pktbuf (AVPacketList **, AVPacket *, AVPacketList **)` ubicada en el fichero `/MPlayer/libavformat/utils.c`.

La Fig. 58 muestra el contenido de esa función destacando las sentencias encargadas de incluir el paquete en la lista. En realidad, la inclusión citada no hace referencia a volver a guardar el contenido del paquete en otro lugar diferentes sino únicamente la dirección de memoria que apunta a ese contenido. De este modo, se tienen acceso a los paquetes generados en la sub-etapa anterior de forma fácil y eficiente.

```
static AVPacket *add_to_pktbuf(AVPacketList **packet_buffer, AVPacket *pkt, AVPacketList **plast_pktl){
    AVPacketList *pktl = av_mallocz(sizeof(AVPacketList));
    if (!pktl)
        return NULL;

    if (*packet_buffer)
        (*plast_pktl)->next = pktl;
    else
        *packet_buffer = pktl;

    /* add the packet in the buffered packet list */
    *plast_pktl = pktl;
    pktl->pkt= *pkt;
    return &pktl->pkt;
}
```

Fig. 58 Sección de código que indexa un paquete a la lista de paquetes (función add_to_pktbuf)

Por tanto, pese a que se tienen referenciados un número de paquetes importante (del orden de cuarenta en función del fichero), la metodología de trabajo de MPlayer en adelante cambia notablemente. Es más, esa copia masiva de paquetes se realiza únicamente en la fase inicial de reconocimiento de *streams*. A partir de ahora, las operaciones se realizarán paquete a paquete. Sin embargo, tal y como se dijo en la introducción de la etapa previa al descodificador, los procesos que se realizan en esta sub-etapa de adaptación sí que se encuentran fuertemente relacionados con los realizados en la sub-etapa de identificación con algunos matices que se verán a continuación.

Ese cambio en la metodología de trabajo del reproductor mencionado líneas atrás comienza por distinguir en qué momento del proceso completo de descodificación de un vídeo tiene lugar, es decir, mientras que la identificación de *streams* se realiza una sola vez al inicio del proceso, esta sub-etapa denominada adaptación del demultiplexor forma parte del bucle principal de descodificación. En concreto, se podría afirmar que el demultiplexor es el suministrador de paquetes, uno a uno, del descodificador. Esta particularidad de trabajar paquete a paquete es muy importante resaltarla ya que en ella se han basado los procedimientos de comunicación e intercambio de información entre procesadores que más adelante se describirán (apartados 4.3.1 y 4.3.2). Por último, antes de analizar las funciones más significativas que permiten llevar a cabo el proceso, es necesario aclarar que en este apartado MPlayer nuevamente vuelve a hacer uso del concepto de paquete. Para diferenciar estos paquetes de los que se encuentran en la lista de paquetes de la fase anterior, se llamará a los nuevos paquetes "paquete_demux" mientras que los paquetes de la lista serán "paquete_lista" evitando equivocaciones.

La idea básica de funcionamiento de esta adaptación es la siguiente: cada llamada que se haga al descodificador para que procese un "paquete_demux" estará precedida de una copia de un paquete de la lista de paquetes ("paquete_lista") a un "paquete_demux", es decir, el demultiplexor irá proporcionado paquetes de uno en uno al descodificador, tal y como ya se ha mencionado en varias ocasiones. En cuanto al tamaño del nuevo "paquete_demux", éste se corresponderá con el tamaño del paquete de la lista de paquetes ("paquete_lista") empleado en la copia.

Además, hay que tener en cuenta que el contenido de la lista de paquetes almacenados en la sub-etapa anterior no es infinito, por lo que una vez que se acaben se volverán a realizar los procesos tanto de copia de datos desde fichero (en bloques de 32 KB) como de generación

e inclusión de "paquetes_lista" en la lista, pero en este caso se hará también de uno en uno. La Fig. 59 representa el comportamiento descrito, diferenciando entre las dos situaciones comentadas. El caso (A) representa la primera situación descrita en la que todavía se tienen paquetes en la lista de paquetes rellenada en la sub-etapa de identificación de *streams*, de modo que se obtiene un "paquete_demux" por cada llamada al descodificador. Por el contrario, en el caso (B) ya se ha vaciado la lista de paquetes, de manera que se vuelve a recurrir a la tarea de copiar información de fichero y generar un paquete que se incluya en la lista de paquetes. Aclarar que solo existe una lista de paquetes, pero para facilitar la comprensión de la figura se han dibujado dos.

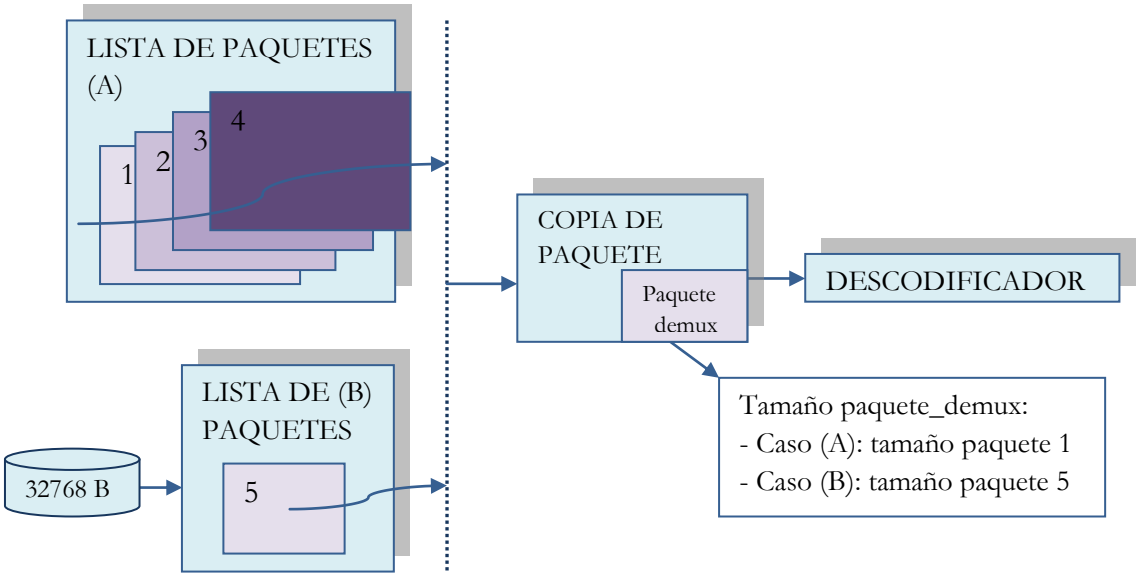

Fig. 59 Modo de obtención de paquetes del demultiplexor

A la vista del funcionamiento en el caso (B), no parece necesario incluir el paquete en la lista ya que cuando el descodificador procese dicho paquete, éste se eliminará y se generará e incluirá en la lista de paquetes otro nuevo a partir de los datos restantes de fichero (32 KB menos el tamaño del paquete procesado) de modo que nunca habrá más de uno en dicha lista. Existiría la posibilidad de modificar esa operación pero tal y como en la introducción del capítulo, ha sido prioritario en todo momento mantener el método de trabajo del reproductor.

Las funciones encargadas de realizar la copia de un paquete de la lista de paquetes a un paquete_demux se muestran a continuación. La función `demux_packet* new_demux_packet (int)` ubicada en el fichero `/MPlayer/libmpdemux/demuxer.h` reserva una zona de memoria de tamaño la longitud del paquete actual y obtiene la dirección de memoria a partir de la cual se copiarán los datos de dicho paquete.

La Fig. 60 muestra el contenido de dicha función destacando aquellas sentencias de importancia.

```
static inline demux_packet_t* new_demux_packet(int len){
  demux_packet_t* dp=(demux_packet_t*)malloc(sizeof(demux_packet_t));
  dp->len=len;
  dp->next=NULL;
  dp->pts=MP_NOPTS_VALUE;
  dp->endpts=MP_NOPTS_VALUE;
  dp->stream_pts = MP_NOPTS_VALUE;
  dp->pos=0;
  dp->flags=0;
  dp->refcount=1;
  dp->master=NULL;
  dp->buffer=NULL;
  if (len > 0 && (dp->buffer = (unsigned char *)malloc(len + MP_INPUT_BUFFER_PADDING_SIZE)))
    memset(dp->buffer + len, 0, 8);
  else
    dp->len = 0;
  return dp;
}
```

Fig. 60 Sección de código que reserva memoria para un paquete del demultiplexor (función new_demux_packet)

Mientras que la copia de datos al denominado paquete_demux se realiza en la función `int demux_lavf_fill_buffer (demuxer_t *, demux_stream_t *)` ubicada en el fichero `/MPlayer/libmpdemux/demux_lavf.c`. Destacar también la eliminación del paquete tras su copia, al tratarse de memoria dinámica.

La Fig. 61 muestra una sección del contenido de dicha función destacando aquellas sentencias de importancia.

```
static int demux_lavf_fill_buffer(demuxer_t *demux, demux_stream_t *dsds){
    if(0/*pkt.destruct == av_destruct_packet*/){
        //ok kids, dont try this at home :)
        dp=malloc(sizeof(demux_packet_t));
        dp->len=pkt.size;
        dp->next=NULL;
        dp->refcount=1;
        dp->master=NULL;
        dp->buffer=pkt.data;
        pkt.destruct= NULL;
    }else{
        dp=new_demux_packet(pkt.size);
        memcpy(dp->buffer, pkt.data, pkt.size*sizeof(unsigned char));
        av_free_packet(&pkt);
```

Fig. 61 Sección de código que copia la información del paquete al demultiplexor (función demux_lavf_fill_buffer)

Con la presentación de las principales funciones que realizan la adaptación del demultiplexor, el descodificador ya está perfectamente preparado para empezar a trabajar, de modo que se pasará a explicar la parte fundamental del desarrollo de este Trabajo Fin de Máster y que mayor esfuerzo ha supuesto debido a la complejidad de cada uno de los elementos que han formado parte del mismo.

4.3 Descodificación

Llegados a este punto, tiene lugar el encaje de cada una de las piezas que han sido comentadas a lo largo de los capítulos 2 y 3 de este trabajo. Al tratarse de una tarea de elevada complejidad, se ha decidido dividir el trabajo en los cuatro elementos empleados más importantes: MPlayer, el descodificador OpenSVC y los módulos DSPLink y CMEM. De cada uno de ellos se detallarán las modificaciones realizadas y las relaciones que guardan

entre sí ya que mientras que en las etapas de entrada y salida del descodificador se han realizado únicamente análisis en profundidad del flujo de datos, en esta ocasión también se llevan a cabo una serie significativa de cambios necesarios para el correcto funcionamiento de la integración MPlayer-OpenSVC.

Antes de entrar en el detalle de alguno de los cuatro elementos mencionados, se vuelve a hacer referencia a la Fig. 47, destacando la zona (color azul) que va a ser descrita a continuacion.

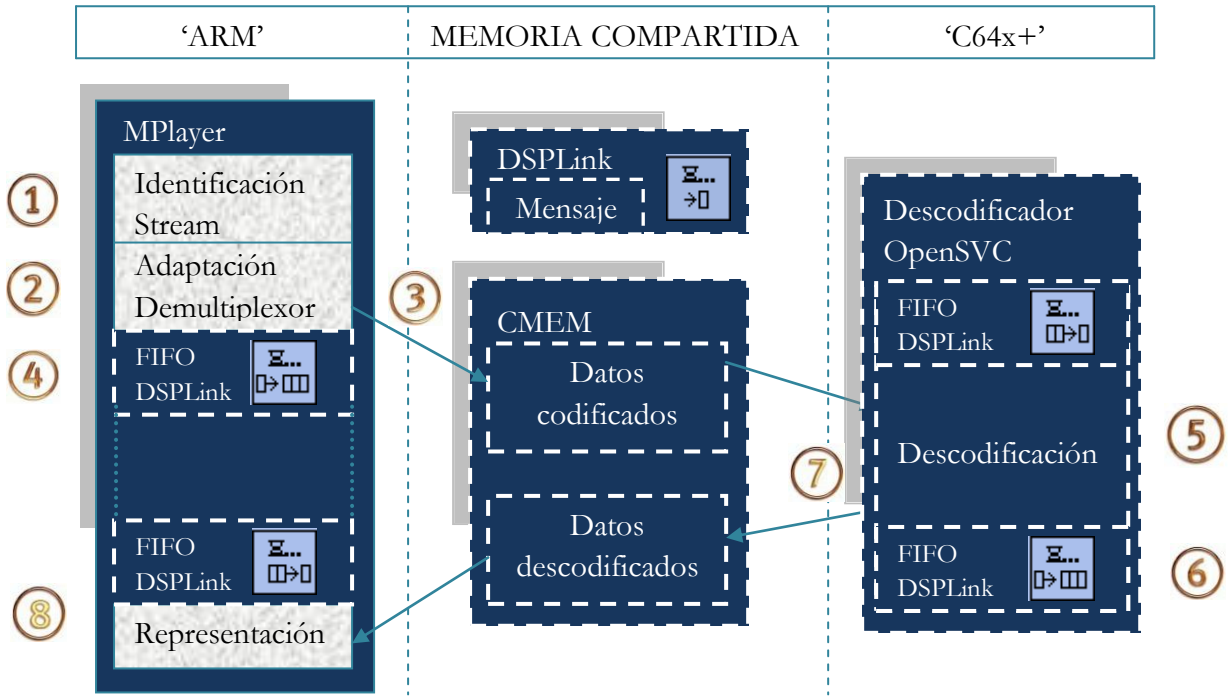

Fig. 47 Diagrama general de la aplicación desarrollada

Dado que tanto MPlayer como el descodificador OpenSVC van a hacer uso de las funcionalidades de los módulos DSPLink y CMEM, se ha optado por analizar en primer lugar los detalles acerca de ambos módulos, de manera que cuando se haga referencia a éstos más adelante, se tenga pleno conocimiento de ellos. Por tanto, se comenzará por el módulo DSPLink y a continuación se explicarán los detalles de CMEM.

4.3.1 DSPLink

El empleo de DSPLink, tal y como se comentó en el apartado 2.3.2.1, persigue dos objetivos fundamentales:

- Sincronización y comunicación entre ambos núcleos del procesador OMAP3530.
- Transferencia de información reducida a través de colas de mensajes.

En el apartado 3.2.2.1 se describió cómo configurar el módulo DSPLink para su posterior empleo, de modo que mediante el procedimiento adecuado se puedan llevar a cabo los objetivos propuestos. En este capítulo se verán cuáles son esos procedimientos y los medios que se emplean para abordar los objetivos mencionados.

DSPLink, dentro de los compontes que proporciona, cuenta con el denominado MSGQ (*Messaging Queue*) [38]. Este componente ofrece un sistema de mensajes basado en colas de tal forma que permite el intercambio de mensajes cortos de longitud variable entre los dos núcleos del procesador OMAP3530. Los mensajes son enviados y recibidos a través de esas colas de mensajes, que por otro lado son bidireccionales, lo que permite reducir el número de ellas.

Este componente está implementado a través de una API cuyas funciones más importantes se presentarán a continuación. De cada una de ellas se comentará brevemente su utilidad y se describirán los parámetros destacando aquellos que hayan tenido más relevancia para la integración MPlayer-OpenSVC. También es importante destacar que las funciones son iguales para ambos núcleos, lo que obligará, como se verá a continuación, a que determinados campos deban ser iguales tanto en el 'ARM' como en el 'C64x+'.

La función `MSGQ_locate` se encarga de localizar de forma síncrona la única cola de mensajes declarada en la aplicación e identificada por un nombre. El nombre de la cola debe estar previamente definido, y debe ser declarado de igual modo tanto por el 'ARM' como por el 'C64x+'. La Fig. 62 muestra ambas definiciones, correspondiendo la parte superior de la imagen al núcleo 'ARM' y la parte inferior al 'C64x+'.

```
/** ============================================================================
 *  @const  SampleGppMsgqName
 *  @desc   Name of the first MSGQ on the GPP.
 *  ============================================================================
 */
STATIC Char8 SampleGppMsgqName [DSP_MAX_STRLEN] = "GPPMSGQ1" ;
/** ============================================================================
 *  @const  SampleDspMsgqName
 *  @desc   Name of the first MSGQ on the DSP.
 *  ============================================================================
 */
STATIC Char8 SampleDspMsgqName [DSP_MAX_STRLEN] = "DSPMSGQ1" ;
/** ============================================================================
 *  @const  DSP_MSGQNAME
 *  @desc   Name of the message queue on the DSP.
 *  ============================================================================
 */
#define DSP_MSGQNAME       "DSPMSGQ1"
/** ============================================================================
 *  @const  GPP_MSGQNAME
 *  @desc   Name of the message queue on the GPP.
 *  ============================================================================
 */
#define GPP_MSGQNAME       "GPPMSGQ1"
```

Fig. 62 Sección de código que define las colas de mensajes en los núcleos del procesador OMAP3530 (arriba: 'ARM' abajo: 'C64x+)

La Tabla 8 muestra información acerca de los parámetros de la función `MSGQ_locate` y una breve descripción de los mismos.

Nombre de la función	Parámetros	Descripción de los parámetros
MSGQ_locate	Pstr queueName	Nombre de la cola de mensajes a localizar.
	MSGQ_Queue * msgqQueue	Ubicación para almacenar el manejador de la cola de mensajes localizada.
	MSGQ_LocateAttrs * attrs	Atributos opcionales para la localización de la cola de mensajes.

Tabla 8 Descripción de la función MSGQ_locate - DSPLink

La función `MSGQ_alloc` se encarga de asignar una zona de memoria dinámica a un mensaje a transferir y a su vez devuelve un puntero al usuario. Esa zona de memoria se ubica dentro de la sección POOL_MEM, de tamaño 832 KB (ver apartado 3.2.1) y debe estar correctamente configurada para que la cola de mensajes funcione correctamente. La Tabla 9 muestra información acerca de sus parámetros y una breve descripción de los mismos.

Nombre de la función	Parámetros	Descripción de los parámetros
MSGQ_alloc	PoolId poolID	Identificador de la pool que será usada para la reserva de memoria del mensaje.
	Uint16 size	Tamaño del mensaje.
	MSGQ_Msg * msg	Ubicación donde se recibirá/escribirá el mensaje reservado.

Tabla 9 Descripción de la función MSGQ_alloc - DSPLink

La función `MSGQ_get` recibe un mensaje en la cola de mensajes especificada en uno de los parámetros de la propia función. Además de éste, la función presenta otro parámetro que permite determinar cuanto tiempo se espera un nuevo mensaje. Este tiempo es configurable mediante macros y existe la opción de realizar esperas bloqueantes (hasta recibir mensaje) o de un tiempo determinado expresado en milisegundos. En este trabajo se han realizado esperas bloqueantes al tratarse de una versión funcional que trate de asegurar el funcionamiento de la aplicación. La Tabla 10 muestra información acerca de sus parámetros y una breve descripción de los mismos.

Nombre de la función	Parámetros	Descripción de los parámetros
MSGQ_get	MSGQ_Queue msgQueue	Manejador a la cola en la cual se recibirá el mensaje.
	Uint32 timeout	Tiempo de espera para el mensaje (en milisegundos).
	MSGQ_Msg * msg	Ubicación donde se recibirá el mensaje.

Tabla 10 Descripción de la función MSGQ_get - DSPLink

La función `MSGQ_put` es la complementaria a la anteriormente descrita `MSGQ_get`. Se encarga de enviar un mensaje en la cola de mensajes especificada en uno de los parámetros de la propia función. La Tabla 11 muestra información acerca de sus parámetros y una breve descripción de los mismos.

Nombre de la función	Parámetros	Descripción de los parámetros
MSGQ_put	MSGQ_Queue msgQueue	Manejador a la cola destino donde se escribirá el mensaje.
	MSGQ_MSG * msg	Puntero al mensaje que ha sido enviado a la cola destino.

Tabla 11 Descripción de la función MSGQ_put - DSPLink

Por último, la función `MSGQ_free` se encarga de liberar la zona de memoria dinámica correspondiente al mensaje ya procesado. La Tabla 12 muestra información acerca de sus parámetros y una breve descripción de los mismos.

Nombre de la función	Parámetros	Descripción de los parámetros
MSGQ_free	MSGQ_MSG * msg	Puntero al mensaje que será liberado.

Tabla 12 Descripción de la función MSGQ_free - DSPLink

A través de las funciones y sus parámetros configurados adecuadamente se consigue el primero de los objetivos propuestos, la sincronización y comunicación entre los dos núcleos del procesador OMAP3530. Sin embargo, para el segundo de los objetivos, el intercambio de información a través de mensajes, falta por describir cuáles son los campos a transferir en un mensaje. Nuevamente es necesario que tanto el 'ARM' como el 'C64x+' implementen la misma estructura, ya que de lo contrario los valores obtenidos por cada núcleo en el mensaje transferido no serán los mismos.

Aunque de momento no sean explicado la mayoría de los campos incluidos en la estructura que se transfiere, la Fig. 63 muestra el contenido de dicha estructura.

```
typedef struct SampleMessage_tag {
    MSGQ_MsgHeader msgHeader     ;
    Uint32         gppWriteAddr  ;              //Dirección física que apunta al final del paquete transferido (dspWriteAddr+size)
    Uint32         dspWriteAddr  ;              //Dirección física que apunta al comienzo del paquete transferido
    Uint32         imagenDSP_Componentes;       //Dirección física que apunta al comienzo de la imagen descodificada
    Uint32         size          ;              //Tamaño del paquete transferido
    Uint32         imagen_disponible ;          //Flag que indica si se ha descodificado una imagen en el 'C64x+'
    Uint32         xdimension ;                 //Dimensión x de la imagen descodificada
    Uint32         ydimension ;                 //Dimensión y de la imagen descodificada
    Uint32         linesize_imagenDSPdesco[3];  //Información de las dimensiones de la imagen con bordes
} SampleMessage ;                               //Estructura del mensaje empleada en la API del componente MSGQ
```

Fig. 63 Campos de la estructura del mensaje

Más adelante (ver apartado 4.3.5) se detallará cada uno de los campos que forman parte de la estructura, ya que involucra tanto al módulo CMEM como al descodificador OpenSVC. También se dejará para apartados posteriores la ubicación y contenido de cada campo por razones similares a las comentadas anteriormente.

4.3.2 CMEM

El empleo de este módulo tiene como objetivo solucionar los inconvenientes planteados por ambos núcleos del procesador OMAP3530 en cuanto a la gestión de memoria tal y como se explicó en el apartado 2.3.2.2. Posteriormente, en el apartado 3.2.2.2 se llevó a cabo la configuración del módulo para disponer de su utilidad y a continuación, al igual que se ha hecho con el módulo DSPLink, se presentarán algunas de las funciones que proporciona la API de CMEM así como otros aspectos significativos referentes al módulo.

A diferencia de DSPLink, CMEM presenta una estructura mucho menor en cuanto a número de funciones y complejidad. A continuación se presentan las usadas en este Trabajo Fin de Máster.

La función `CMEM_init` se encarga de inicializar el módulo correctamente. Es en este punto donde se detectan configuraciones erróneas como por ejemplo la generación del módulo para una versión del *kernel* Linux embebido diferente (ver 2.3.2). La Tabla 13 muestra información acerca de sus parámetros y una breve descripción de los mismos.

Nombre de la función	Parámetros	Descripción de los parámetros
CMEM_init	Sin parámetros	-

Tabla 13 Descripción de la función CMEM_init - CMEM

La función `CMEM_alloc` se encarga de asignar espacios de memoria del tamaño especificado por el usuario en uno de los parámetros de la función. Como puede existir más de una *pool* declarada, dos en el caso de este trabajo, esta función proporciona el espacio de memoria en aquella *pool* que mejor se ajuste al tamaño demandado (parámetro *size* de la función).

Del otro parámetro de la función se quiere destacar la importancia que tiene su configuración, ya que permite ubicar el espacio de memoria asignado bien en alguna de las *pools* declaradas o bien en memoria dinámica (*heap*). El desconocimiento de dónde se ubiquen los datos, tal y como sucedió en el desarrollo de este trabajo, puede dar lugar a que los datos intercambiados entre los núcleos del procesador OMAP3530 se copien en memoria dinámica correctamente pero hasta una determinada cantidad de datos, en concreto el tamaño reservado para la *heap*. Por tanto, es de significativa importancia indicar que los datos se guarden en las *pools* a través del segundo parámetro de esta función.

Para evitar posibles errores, se propone visualizar los mensajes que proporciona el módulo en depuración, ya que genera información acerca de en qué lugar se han realizado las asignaciones de memoria (*pools* o *heap*). La Fig. 64 muestra esa información resaltando aquellos campos mencionados.

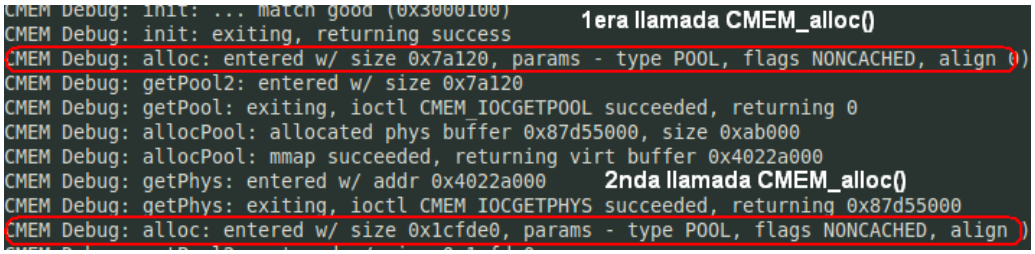

Fig. 64 Asignación de las dos *pools* demandadas en CMEM

Volviendo a la función, la Tabla 14 muestra información acerca de sus parámetros y una breve descripción de los mismos.

Nombre de la función	Parámetros	Descripción de los parámetros
CMEM_alloc	Size_t size	Tamaño del espacio de memoria a reservar.
	CMEM_Alloc_Params *params	Parámetros de la asignación (ubicación, uso de memoria caché)

Tabla 14 Descripción de la función CMEM_alloc - CMEM

Por último, la tercera función, `CMEM_getPhys` permite obtener la dirección física al inicio del espacio de memoria reservado en la función anterior. La Tabla 15 muestra información acerca de sus parámetros y una breve descripción de los mismos.

Nombre de la función	Parámetros	Descripción de los parámetros
CMEM_getPhys	Void * ptr	Puntero al espacio de memoria reservado

Tabla 15 Descripción de la función CMEM_getPhys - CMEM

La problemática de direcciones virtuales y físicas en los núcleos 'ARM' y 'C64x+' respectivamente hace necesario el empleo de herramientas que "traduzcan" esas direcciones para poder trabajar sobre un único espacio de memoria compartida (ver apartado 2.3.1). Nuevamente se vuelve a hacer uso de la información que proporciona CMEM en depuración para comprobar cuáles son las direcciones virtuales donde se han realizado las reservas de memoria, sus correspondientes direcciones físicas tras la ejecución de la función `CMEM_getPhys` y los tamaños de esas reservas indicados en la función `CMEM_alloc`. La Fig. 65 recoge esta información a la que se le han añadido comentarios que ayudan a comprender mejor el funcionamiento del módulo.

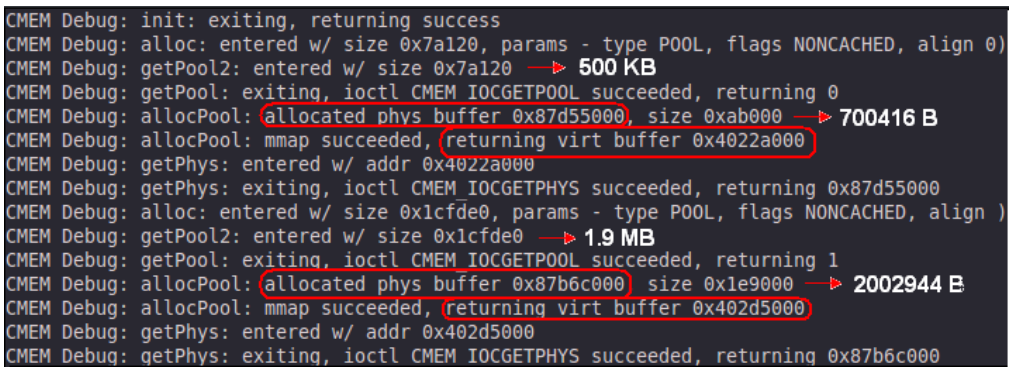

Fig. 65 Configuración del módulo CMEM a través del terminal

El número de bloques demandados es de dos, como se comentó en el apartado 3.2.2.2 y en cuanto a sus tamaños, el primero de ellos tiene un tamaño de 500 KB, valor más que suficiente para la copia de paquetes. Además, al realizarse la copia de paquetes de uno en uno, este valor asegura aún más el correcto funcionamiento ya que durante el desarrollo del trabajo no se han contemplado paquetes de tamaño superior a 100 KB. El tamaño del segundo bloque estaba condicionado por el tamaño de las imágenes descodificadas (ver apartado 3.2.2.2) y se ha optado por reservar 1.9 MB para evitar problemas de espacio y ofrecer facilidades para futuras ampliaciones de la aplicación. En ambas *pools* se ha preferido no ajustar al máximo las cantidades demandadas (500 KB y 1.9 MB) respecto de las reservadas en la configuración del módulo (700 KB y 2 MB) dejando un margen de seguridad.

Sintetizando la información de la Fig. 65 se ha confeccionado una tabla (ver Tabla 16) a modo resumen para mostrar la configuración final realizada.

Número de Pool	Tamaño de la pool		Dirección de inicio	
	Conf. Módulo	Demandada	Virtual	Física
1	700 KB	500 KB	0x4022A000	0x87D55000
2	2 MB	1.9 MB	0x402D5000	0x87B6C000

Tabla 16 Síntesis de la configuración realizada en CMEM

Por último, comentar que se verá en el apartado 4.3.5 al igual que se comentó para DSPLink, la ubicación de las funciones de CMEM y algunas particularidades más del módulo. Es preferible esta organización ya que aún faltan por ver las modificaciones realizadas tanto en el reproductor como en el descodificador, de tal forma que una vez contempladas todas ellas, será más sencillo situar la ubicación de las funciones de ambos módulos y las relaciones que guardan junto al resto de elementos de la aplicación.

4.3.3 Modificaciones en el reproductor MPlayer

En este apartado se describirán únicamente los cambios realizados en el reproductor necesarios para poder emplear las API tanto de DSPLink como de CMEM comentadas con anterioridad así como la creación de un fichero de encabezado y una serie de modificaciones en algunas estructuras propias de MPlayer para transferir la información de variables pertenecientes al modulo CMEM y la imagen descodificada a través de diferentes funciones del reproductor.

En primer lugar, para poder emplear las funciones de ambos módulos ha sido necesario incluir sus librerías correspondientes dentro de la estructura de MPlayer, en concreto modificando el fichero *makefile* que emplea el reproductor para realizar los procesos de compilación, enlazado y construcción del ejecutable. Este fichero, dado el elevado número de ellos y de distinto tipo que conforman el reproductor, ha sido analizado para conocer exactamente dónde incluir los siguientes elementos:

- Librería de DSPLink: `dsplink.a` (ubicada en `/MPlayer/DSPLink/gpp/BUILD/EXPORT/DEBUG/`).
- Librería de CMEM: `cmemd.a470uC` (ubicada en `/MPlayer /DSPLink /cmem/lib/`).
- Ficheros de encabezado de ambos módulos (**.h*)

El fichero *makefile* presenta una organización densa en cuanto a contenido y compleja en cuanto a comprensión del mismo, ya que está implementado mediante macros muy similares todas ellas. Además, analizando el proceso de compilación y enlazado del mismo, se ha observado que existen multitud de pequeños grupos de ficheros fuente y de encabezado que forman por sí solos una librería. Ésto significa que existen librerías agrupadas por funcionalidad: una librería para el estándar H.264/SVC, otra librería para un determinado tipo de demultiplexor, otra para el controlador de salida de vídeo, etc. Este tipo de organización lleva a que si se desea emplear, por ejemplo, una función perteneciente al módulo DSPLink en un fichero que pertenece a la librería 'C', es necesario localizar dónde están definidos los ficheros fuente de esa librería 'C' en el fichero *makefile* e

incluir junto a ellos los ficheros fuente donde se defina la función del módulo DSPLink. La Fig. 66 refleja este comportamiento descrito con algunos comentarios adicionales.

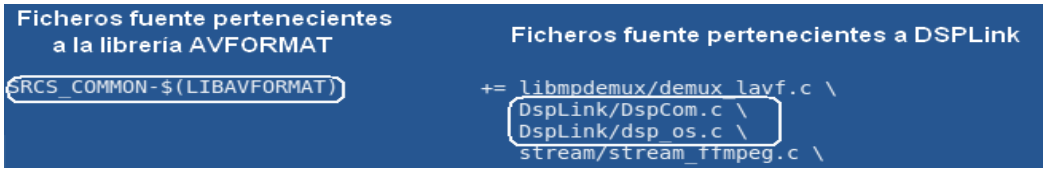

Fig. 66 Inclusión de ficheros fuente en la librería AVFORMAT

Mediante la figura presentada puede deducirse alguna particularidad más acerca del funcionamiento del fichero *makefile* de MPlayer. Por ejemplo, que la función perteneciente al módulo DSPLink se va a emplear en alguno de los ficheros fuente que aparecen bajo la librería LIBAVFORMAT, en concreto se hará en el fichero `demux_lavf.c`, como más adelante se verá. En consecuencia, ha sido necesario añadir a esa librería los ficheros fuente donde se encuentra el contenido de esa función, que son los que se muestran resaltados en la figura (`DSPCom.c` y `dsp_os.c`). A continuación será necesario ejecutar la instrucción *make* en una terminal de consola para actualizar la librería y a su vez el propio ejecutable del reproductor.

Respecto a las modificaciones realizadas en el *makefile* para trabajar con las API de DSPLink y CMEM, la Fig. 67 muestra la configuración realizada en el propio fichero.

```
COMMON_LIBS-$(LIBAVFORMAT_A)      += DspLink/gpp/BUILD/EXPORT/DEBUG/dsplink.a
COMMON_LIBS-$(LIBAVFORMAT_A)      += DspLink/cmem/lib/cmemd.a470uC

DIRS =  . \
        DspLink/cmem/include \
        DspLink/gpp/inc/usr \
```

Fig. 67 Inclusión de librerías de DSPLink y CMEM en MPlayer

Por otro lado, se ha creado un fichero de encabezado denominado `libreríaOHA.h` que se presenta a continuación (ver Fig. 68) y cuyo objetivo es almacenar los campos tanto de la imagen descodificada como de las *pools* declaradas en CMEM para tener siempre disponible la información en el núcleo 'ARM'.

libreriaOHA.h

```
typedef struct {
  unsigned int * addr_poolcmem1;        //Puntero al comienzo de la primera pool reservada en CMEM (dirección virtual)
  unsigned int * addr_poolcmem2;        //Puntero al comienzo de la segunda pool reservada en CMEM (dirección virtual)
  size_t size_buffer1CMEM;              //Tamaño de la primera pool reservada en CMEM
  size_t size_buffer2CMEM;              //Tamaño de la segunda pool reservada en CMEM
  unsigned int xdim_Imagen;             //Dimensión 'x' de la imagen descodificada por el núcleo 'C64x+'
  unsigned int ydim_Imagen;             //Dimensión 'y' de la imagen descodificada por el núcleo 'C64x+'
  unsigned int flag_imagen_descodificada;       //Flag de aviso de imagen descodificada
  unsigned int linesize_imagenDSP[3];   //Información de los bordes de la imagen descodificada
}infoCMEM_t_imagen;
```

Fig. 68 Visualización del fichero de encabezado libreríaOHA.h

Además, se ha optado por no modificar el prototipo de ninguna función perteneciente a MPlayer debido a la alta dependencia que presenta el reproductor entre sus propias funciones. En consecuencia, cuando ha sido necesario el paso de parámetros significativos entre funciones, se han añadido esos parámetros a algunas de las estructuras ya existentes del reproductor (ver apartado 4.4.2), de manera que se dispone de la información de un modo rápido y de sencilla configuración. En concreto, se han añadido los campos que se

presentan a continuación en las dos estructuras de datos: `sh_video_t` y `AVCodecContext` ubicadas en los ficheros `/MPlayer/libmpdemux/stheader.h` y `/MPlayer/libavcodec/avcodec.h` respectivamente.

- `Unsigned int * imagen_datos_NombreEstructura`
- `Int dimension_x_Nombre_Estructura`
- `Int dimension_y_Nombre_Estructura`
- `Int flag_imagenDSP_NombreEstructura`

El significado, empleo y la actualización de estos campos se verá en el apartado 4.4.2, una vez que sea analizado el último de los principales integrantes de la aplicación, el descodificador vídeo escalable OpenSVC.

4.3.4 Modificaciones en el descodificador OpenSVC

Como se mencionó en el capítulo 1 de este Trabajo Fin de Máster, ya se contaba con una versión del descodificador de vídeo escalable OpenSVC operativa y optimizada [2][3] para el núcleo 'C64x+'. Por este motivo, la integración de herramientas como DSPLink no ha sido necesario realizarla al encontrarse ya implementada. Sin embargo, el método de trabajo presentado en esa versión del descodificador no ha podido ser mantenido debido a problemas de compatibilidad con el funcionamiento de MPlayer. A continuación se detallarán los cambios realizados en la versión actual con el fin de clarificar cuál es el nuevo método de trabajo y qué diferencias existen con el método anterior.

Recordando la etapa previa al descodificador comentada en el primer apartado de este capítulo, éste va a procesar por cada llamada que se le haga un único paquete de longitud variable cuyo tamaño no superará en ningún caso el tamaño de la primera *pool* de CMEM (500 KB). Es más, durante el desarrollo del trabajo realizado no se han procesado paquetes de tamaño superior a 100 KB, de modo que se va a trabajar directamente sobre el espacio de memoria compartida, evitando realizar una nueva copia de datos hacia la memoria externa del núcleo 'C64x+', lo que implicaría mayor número de ciclos de reloj y en definitiva más lentitud. En este punto ya se encuentra la primera diferencia con el método anterior, ya que en esa versión se copiaban bloques completos de tamaño muy superior, del orden de 400 KB, de manera que se copiaban los datos a memoria externa del procesador digital de señal para no bloquear la zona de memoria compartida durante el procesamiento de ese bloque.

Dejando a un lado las funciones pertenecientes a DSPLink (`MSGQ_get` y `MSGQ_put`), cuya función y ubicación dentro del descodificador se comentarán más adelante (ver apartado 4.3.5), y las tareas de inicialización del mismo, el paquete procesado contiene un número exacto de unidades NAL tal y como se citó en el apartado 4.2.1. Para el análisis de las NAL del paquete, ha sido necesario modificar la estructura del descodificador, realizando cambios en un conjunto de funciones que serán presentadas a continuación junto a otra serie de particularidades llevadas a cabo.

Una primera aproximación de cómo se procesan las diferentes unidades NAL contenidas en un paquete se presenta en la Fig. 69.

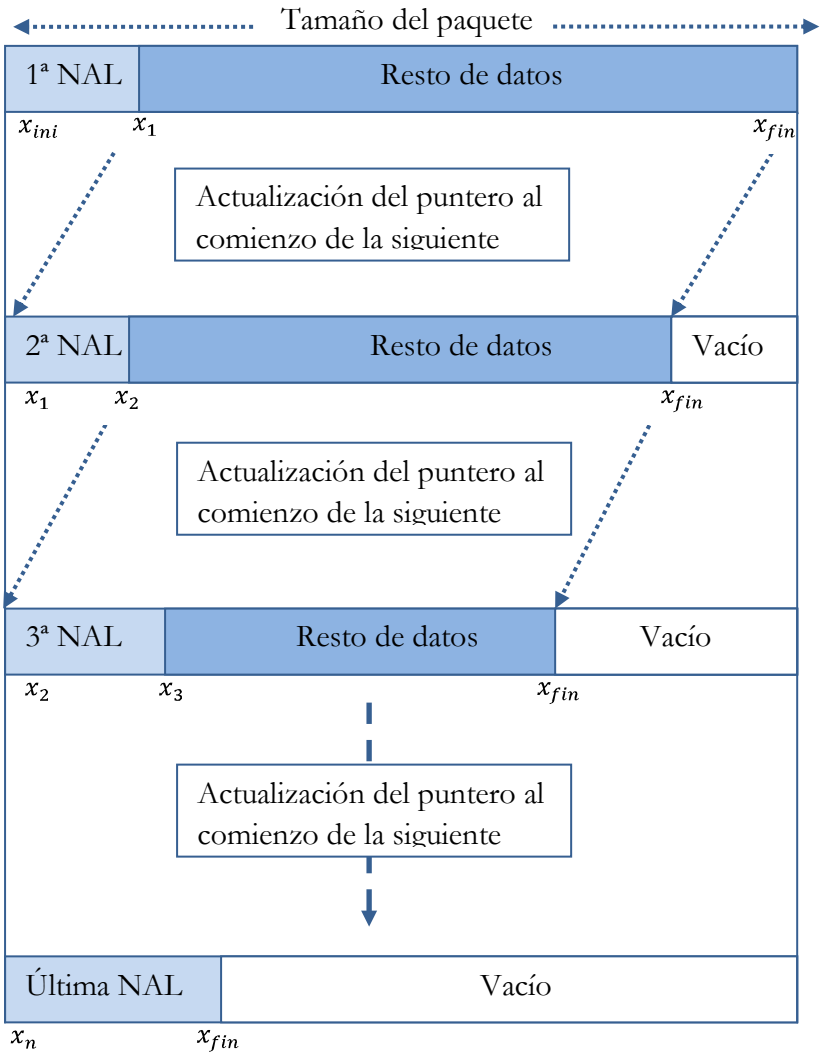

Fig. 69 Obtención de las diferentes NAL del paquete procesado

La imagen trata de explicar como la dirección de memoria que apunta al comienzo de la cada NAL se ve incrementada por una cantidad igual al tamaño de la NAL procesada, de tal forma que para el primero de los casos, el valor de x_1 será igual a x_{ini} más el tamaño de esa NAL (1ª); para el segundo, el valor de x_2 será igual a x_1 más el tamaño de la NAL procesada (2ª) y así sucesivamente hasta procesar la última NAL que contiene el paquete. El incremento de la dirección de memoria para cada NAL puede implementarse debido a que la estructura global del descodificador gira en torno a un proceso iterativo.

Este procesamiento descrito implica llevar un control de los *bytes* procesados por cada paquete recibido, así como conocer el tamaño exacto de cada NAL para actualizar el puntero tantas direcciones de memoria como *bytes* tenga la NAL procesada. Un detalle importante a destacar de forma breve es los campos por los que está formada una unidad NAL [2]: desde el punto de vista del trabajo desarrollado, está formado por dos: un identificador de comienzo de NAL (*offset*) y los propios datos. El primero de los campos es constante y está formado por cuatro octetos cuyos valores son 00 00 00 01. En consecuencia, todas las unidades NAL comenzarán por esta secuencia binaria, por lo que un modo rápido de comprobar si realmente se están procesando las NAL de forma

completa es imprimir por consola los cuatro primeros octetos en cada iteración del bucle de descodificación. La Fig. 70 muestra las instrucciones para llevarlo a cabo.

```
printf("\nPuntero CMEM móvil a los datos:0x%x\n",(unsigned long)writeBuf);
printf("\nImprimir los siguiente cuatro elementos (cabecera NAL 0 0 0 1):%d,%d,%d,%d\n",
                                        writeBuf[0],writeBuf[1],writeBuf[2],writeBuf[3]);
```

Fig. 70 Sección de código que imprime la cabecera de la NAL a procesar (función TSKPROCESADO_execute)

Una vez corroborado que las NAL se procesan con éxito, es preferible eliminar los mensajes para acelerar la descodificación.

A nivel de cambios realizados en el código de las funciones del descodificador, han sido tres las estudiadas y modificadas para conseguir los objetivos de control de *bytes* por paquete y obtención del tamaño de cada NAL:

- `Int TSKPROCESADO_execute (TSKPROCESADO_TransferInfo *)` ubicada en el fichero `tskProcesado.c`
- `Int GetNalBytesInNalunit (unsigned char *, unsigned char *, int *, CONFIGSVC *, int *, int)` ubicada en el fichero `nal_unit_header_svc_extension.c`
- `Int search_start_code_SVC_OHA (unsigned char *, int *, int *, int)` ubicada en el fichero `Nal.c`

Las dos primeras funciones ya pertenecían al descodificador, mientras que la última se ha optado por duplicarla y modificar su nombre ya que en la función original (`search_start_code_SVC`) se realizaron cambios en su prototipo que afectaban a numerosos ficheros, de modo que se decidió por crear una nueva función basándose en la original para evitar fallos en tiempo de compilación. Por otro lado, en esta ocasión no se van a mostrar secciones de código pertenecientes a las funciones descritas ya que por su estructura y el alto grado de dependencia entre ellas hace dificil la comprensión de las mismas. Se ha optado por realizar un diagrama de flujo representando el proceso completo de descodificación. Para ello, antes de presentar el diagrama, se va a describir el resultado final de la descodificación, esto es, cómo se guardan las imágenes descodificadas en memoria, en concreto, en la segunda *pool* de CMEM definida para este tipo de información.

En referencia a la copia de datos, es importante destacar que el descodificador alojado en el núcleo 'C64x+' únicamente comienza el proceso de guardar imagen cuando ha procesado todas las NAL de un paquete (en caso de que ese paquete contenga una imagen). Esto se debe a la gestión que hace MPlayer en la generación de paquetes (ver apartado 4.2.1), de manera que no se interrumpe el análisis de un paquete hasta que no se terminen de procesar todas sus unidades NAL. Cuando se produzca ese fin de procesado, entonces se extrae la información de la imagen en sus tres componentes de forma independiente: 'Y' (luminancia) y 'U' y 'V' (crominancias); y también sus dimensiones. La Fig. 71 muestra la sección de código perteneciente a la función `TSKPROCESADO_execute` donde se llevan a cabo estas operaciones.

```
uchar *Y = Display_1_Extract_Image_Y_o + 8;                  //Copia del puntero al comienzo de los datos de luminancia
uchar *U = Y + (XDIM + 32) * YDIM;                           //Copia del puntero al comienzo de los datos de la 1era crominancia
uchar *V = U + (XDIM + 32) * YDIM/4 ;                        //Copia del puntero al comienzo de los datos de la 2nda crominancia
int XDIM = ((unsigned int *) Display_1_Extract_Image_Y_o)[0]; //Dimensión x de la imagen descodificada
int YDIM = ((unsigned int *) Display_1_Extract_Image_Y_o)[1]; //Dimensión y de la imagen descodificada
```

Fig. 71 Sección de código que guarda la información de la imagen descodificada (función TSKPROCESADO_execute)

Esta copia de punteros no es suficiente, ya que esa información necesita ser transferida al núcleo 'ARM' para que sea representada en el monitor. Por lo tanto, es necesario copiar no el puntero, sino el contenido de cada componente de la imagen a una zona visible por los dos núcleos del procesador OMAP3530, la segunda de las *pools* declaradas por el módulo CMEM. Esa copia se realiza en la función `void yuv_save (unsigned char **, int *, int, int, char *, char *)` ubicada en el fichero `argvParser.c` (ver Fig. 72).

```
void yuv_save(unsigned char **buf, int *wrap, int xsize, int ysize, char *filename,char * punteroImagen)
{
    int i;

    //Copia en memoria compartida de la info de la imagen (Y,U,V)
    for(i=0; i<ysize; i++)
    {
        #ifndef SAVE_MEM_COMP
            fwrite(buf[0] + i * wrap[0], 1, xsize, f);

        #else
        memcpy(punteroImagen,buf[0] + i * wrap[0],xsize+32);//Añadido +32 (último MB de la línea actual…
        //punteroImagen=punteroImagen+xsize;                //…y primer MB de la siguiente)
        punteroImagen=punteroImagen+(xsize+32);             //Se actualiza puntero (línea horizontal + borde).
        #endif
    }

    for(i=0; i<ysize/2; i++)
    {
        #ifndef SAVE_MEM_COMP
            fwrite(buf[1] + i * wrap[1], 1, xsize/2, f);
        #else
        memcpy(punteroImagen,buf[1] + i * wrap[1], (xsize+32)/2);//Añadido +32 (último MB de la línea actual…
        //punteroImagen=punteroImagen+(xsize/2);                 //…y primer MB de la siguiente)
        punteroImagen=punteroImagen+((xsize+32)/2);              //Se actualiza puntero (línea horizontal + borde).
        #endif
    }

    for(i=0; i<ysize/2; i++)
    {
        #ifndef SAVE_MEM_COMP
            fwrite(buf[2] + i * wrap[2], 1, xsize/2, f);
        #else
        memcpy(punteroImagen,buf[2] + i * wrap[2],(xsize+32)/2);//Añadido +32 (último MB de la línea actual…
        //punteroImagen=punteroImagen+(xsize/2);                //…y primer MB de la siguiente)
        punteroImagen=punteroImagen+((xsize+32)/2);             //Se actualiza puntero (línea horizontal + borde).
        #endif
    }
}
```

Fig. 72 Sección de código que copia la información de la imagen en CMEM (función yuv_save)

La función representada en la imagen presenta un conjunto de parámetros de entrada de los cuales hay que destacar un doble puntero denominado *buf* que contiene a su vez los punteros pertenecientes a las componentes Y, U y V de la imagen, los dos parámetros que contienen las dimensiones de la imagen (ancho x largo) y por último, el puntero denominado *punteroImagen*, que referencia al inicio de la segunda *pool* de CMEM. Como se verá en el capítulo posterior (ver apartado 5.2.6), se ha optado por desarrollar una estructura de compilación condicionada para el salvado de imágenes a disco, pudiendo hacer cambios rápidamente y comprobar si realmente la descodificación se realiza de forma correcta.

Respecto a la copia de cada componente en memoria compartida, se ha empleado un proceso iterativo en el cual es importante resaltar una característica del estándar H.264/SVC: la predicción más allá de los límites de la imagen. Este estándar de codificación de vídeo permite realizar predicciones de tamaño macrobloque (16x16) en los bordes de la imagen, por lo que es necesario tener en cuenta esta particularidad en el proceso de copia de la imagen. En concreto, la Fig. 72 muestra como tanto los *bytes* copiados como la actualización del puntero tras la copia ven incrementados su valor en treinta y dos *bytes*, que se corresponden con el último macrobloque de la fila *n* (borde derecho) y con el primer macrobloque de la fila *n+1* (borde izquierdo). Esta copia particular de la imagen supone una novedad respecto al método de trabajo anterior, ya que en este último, se omitía la información del borde.

Una vez descrito el nuevo método de trabajo, se presenta un diagrama a modo de síntesis (ver Fig. 73) para concluir este apartado perteneciente a las modificaciones realizadas en el descodificador OpenSVC.

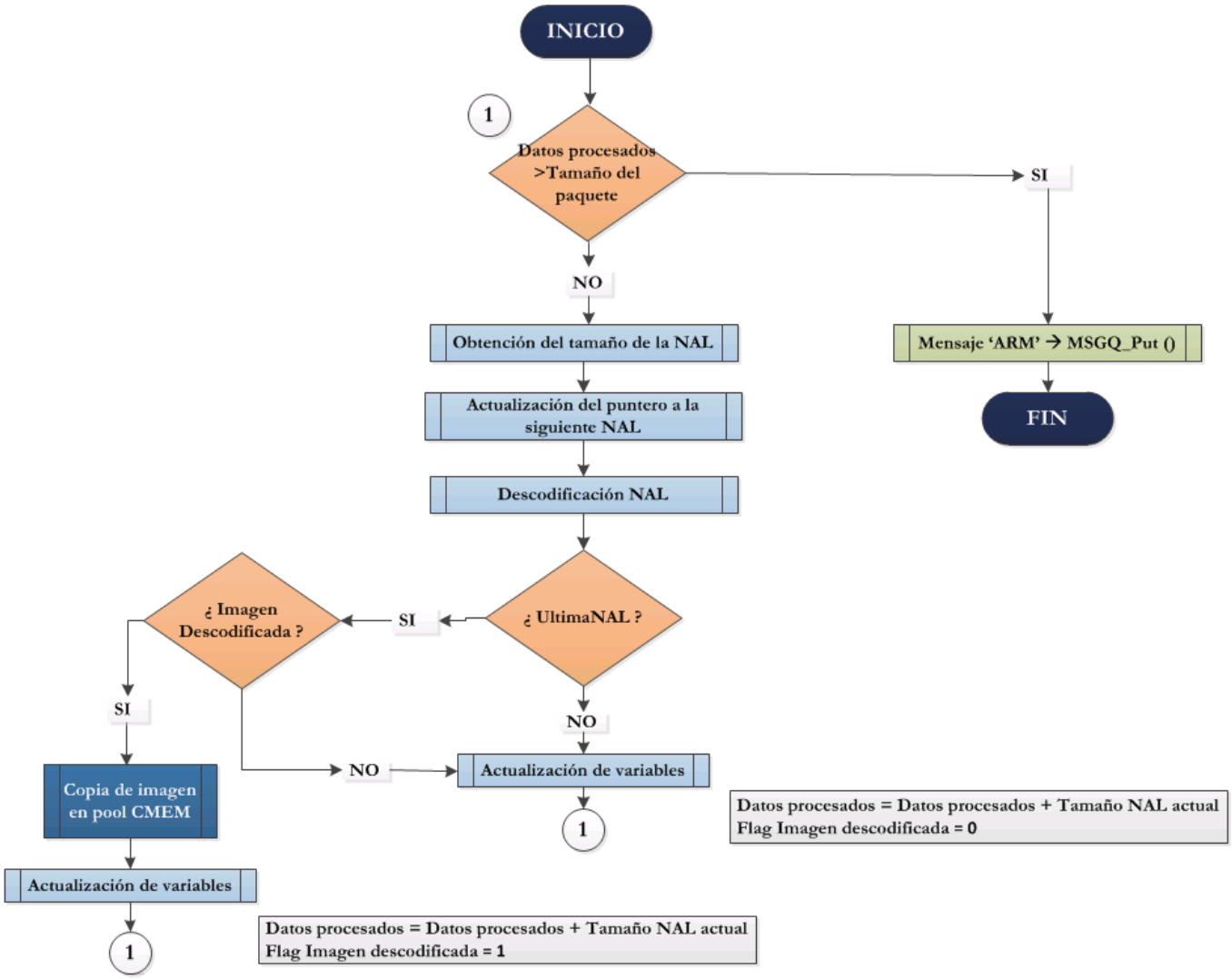

Fig. 73 Diagrama de flujo del método de trabajo del descodificador OpenSVC

Con la descripción de las modificaciones realizadas en el descodificador OpenSVC se termina la presentación de los cambios realizados en los cuatro elementos principales de este Trabajo Fin Máster: los módulos DSPLink y CMEM, el reproductor multimedia MPlayer y el descodificador OpenSVC recién descrito. En consecuencia, el siguiente apartado describirá la secuencia de operaciones realizada para llevar a cabo la integración teniendo en cuenta los cuatro elementos. Para ello, se hará uso nuevamente de diagramas de flujo aclaratorios así como de secciones de código referenciando tanto el nombre de la función como su ubicación dentro de la estructura bien del reproductor o bien del descodificador.

4.3.5 Integración reproductor - descodificador

La integración del reproductor MPlayer alojado en el núcleo 'ARM' con el descodificador OpenSVC alojado 'C64x+' sigue un proceso secuencial. Esto quiere decir que en ningún momento los dos núcleos del procesador OMAP3530 se encuentran en ejecución simúltaneamente ya que se trata de una primera versión funcional, en la que no se ha realizado ninguna optimización por el momento. Sin embargo, como se verá en el capítulo posterior (ver apartado 5.3), sí que se ha conseguido reducir la carga computacional del núcleo 'ARM' realizando la descodificación en el núcleo 'C64x+' respecto a la versión previa a este trabajo donde el proceso de descodificación se realizaba también en el procesador de propósito general.

Por tanto, a continuación se realizará una descripción detallada de las operaciones realizadas siguiendo el orden secuencial mencionado. Dicha descripción irá acompañada de información adicional como secciones de código comentadas, ubicación de dichas secciones en sus respectivos ficheros y otras aclaraciones. Además, se proporcionará un diagrama de flujo completo de la integración para finalizar este apartado.

Como se ha comentado en repetidas ocasiones, el descodificador OpenSVC irá recibiendo paquetes de uno en uno y dichos paquetes se alojarán en la primera *pool* definida por CMEM, lo que implica que desde el núcleo 'ARM' se debe realizar la copia del paquete en esa zona de memoria compartida. Esta operación tiene lugar en la función `int demux_lavf_fill_buffer (demuxer_t *, demux_stream_t *)` ubicada en el fichero `/MPlayer/libmpdemux/demux_lavf.c`. La Fig. 74 muestra la sección de código donde se realiza dicha copia.

```
//Copia del paquete demuxer al inicio de la primera pool CMEM
memcpy(addr_virtual1, dp->buffer, pkt.size*sizeof(unsigned char));
```

Fig. 74 Sección de código que copia el paquete a procesar en CMEM (función demux_lavf_fill_buffer)

La figura muestra como los datos contenidos en el campo *buffer*, cuyo número de *bytes* es igual al tamaño del paquete, son copiados a partir de la dirección virtual *addr_virtual1*, la cual apunta al inicio de la primera *pool* definidida por el módulo CMEM. La ubicación de los datos así como el tamaño de éstos es necesario comunicársela al descodificador alojado en el núcleo 'C64x+' para que lleve a cabo su procesamiento. Sin embargo, previo a este paso, es necesario obtener las direcciones físicas correspondientes a las direcciones virtuales

que apuntan al inicio de las *pools* con el fin de que el 'C64x+' pueda interpretarlas correctamente. La conversión de esas direcciones virtuales se realiza en la misma función donde se lleva a cabo la copia del paquete (`demux_lavf_fill_buffer`) y en concreto a través de la función `CMEM_getPhys` (ver apartado 4.3.2). La Fig. 75 muestra la sección de código donde se realiza dicha conversión.

```
// Find out and print the physical address of this buffer
    unsigned long addr_fisica1=CMEM_getPhys(addr_virtual1);
if (addr_fisica1 == 0)
{
    printf(stderr, "Failed to get physical address of buffer 0x%x\n",(unsigned int) addr_fisica1 );
    CMEM_exit();
}

     unsigned long addr_fisica2=CMEM_getPhys(addr_virtual2);
if (addr_fisica2 == 0)
{
    printf(stderr, "Failed to get physical address of buffer 0x%x\n",(unsigned int) addr_fisica2 );
    CMEM_exit();
}
```

Fig. 75 Obtención de las direcciones físicas de las *pools* CMEM (función demux_lavf_fill_buffer)

Una vez obtenidas las direcciones físicas de ambas *pools*, el siguiente paso es comunicar esa información al descodificador en un mensaje a través de la cola de mensajes proporcionada por DSPLink. Para ello, es necesario previamente rellenar los campos de la estructura del mensaje, tal y como se presentó en la Fig. 63 (ver apartado 4.3.1), la cual vuelve a presentarse para mayor comodidad.

```
typedef struct SampleMessage_tag {
    MSGQ_MsgHeader msgHeader     ;
    Uint32         gppWriteAddr  ;          //Dirección física que apunta al final del paquete transferido (dspWriteAddr+size)
    Uint32         dspWriteAddr  ;          //Dirección física que apunta al comienzo del paquete transferido
    Uint32         imagenDSP_Componentes;   //Dirección física que apunta al comienzo de la imagen descodificada
    Uint32         size          ;          //Tamaño del paquete transferido
    Uint32         imagen_disponible ;      //Flag que indica si se ha descodificado una imagen en el 'C64x+'
    Uint32         xdimension ;             //Dimensión x de la imagen descodificada
    Uint32         ydimension ;             //Dimensión y de la imagen descodificada
    Uint32         linesize_imagenDSPdesco[3]; //Información de las dimensiones de la imagen con bordes
} SampleMessage ;                           //Estructura del mensaje empleada en la API del componente MSGQ
```

Fig. 63 Campos de la estructura del mensaje

En este punto sí que deberían comprenderse mucho mejor los campos que contiene el mensaje, apoyándose en los comentarios de cada uno de ellos. Para que el descodificador conozca toda la información relativa al paquete es necesario escribir en algunos campos de esa estructura. Esta operación tiene lugar en la función `DSP_Execute (Uint32, Uint32, Uint8, infoCMEM_t_imagen)` ubicada en el fichero `/MPlayer/DSPLink/DSPCom.c`. La Fig. 76 muestra la sección de código donde se realizan las modificaciones.

```
//Relleno de los campos de la estructura que se enviará al 'C64x+'
msg->gppWriteAddr  = dspAddr2;              //Puntero al final del paquete a procesar
msg->dspWriteAddr  = dspAddr1;              //Puntero al inicio de la primera pool de cmem (paquete_demux)
msg->imagenDSP_Componentes = cmem_imagenDSP; //Puntero al inicio de la segunda pool de cmem (imagenes descodificada)
msg->size          = size_packet ;          //Tamaño del paquete
```

Fig. 76 Sección de código que rellena la estructura del mensaje – núcleo 'ARM' (función DSPCom.c)

Con la información copiada en la estructura del mensaje, ambos núcleos ya están listos para el intercambio de información: por un lado, el núcleo 'ARM' enviará un mensaje a través de

la función `MSGQ_put` y se quedará esperando mediante la función `MSGQ_get` la respuesta del descodificador. Se ha configurado el parámetro que determina el tiempo de espera del núcleo 'ARM' para que no siga la ejecución del código hasta recibir respuesta del descodificador (macro `WAIT_FOREVER` o `SYS_FOREVER` dependiendo del núcleo). Por otro lado, el núcleo 'C64x+' estará esperando mediante la función `MSGQ_get` la llegada de un mensaje a su cola de mensajes, momento a partir del cual el descodificador iniciará su ejecución.

Esa ejecución comenzará por invalidar tantos datos de la caché del procesador OMAP3530 como *bytes* tenga el anterior paquete procesado, ya que, como se describirá en el capítulo posterior con más detalle (ver apartado 5.2.5), ha habido problemas con la caché que provocaban que los datos del paquete que leía el 'C64x+' fueran distintos a los que escribía el 'ARM'. Mediante esa invalidación se consiguió solucionar el problema, de modo que es la primera operación que se realiza. A continuación, se copia la información del mensaje recibido en variables locales ya que el propio mensaje y su contenido están alojados en memoria dinámica de modo que se evitan posibles copias de valores erróneos o tardios. La Fig. 77 muestra tanto esa inhabilitación de la caché como la copia de los campos del mensaje a variables locales.

```
HAL_cacheInv ((Ptr) copia_1poolcmem,size); //Se actualiza  la  caché  pasando de los datos que tiene
status = MSGQ_get (info->dspMsgqQueue, (MSGQ_Msg *) &msg, SYS_FOREVER) ;
if (status == SYS_OK)
{
    writeBuf = (Char *) msg->dspWriteAddr ;          //Recibe la dirección inicial de los datos enviados
    readBuf  = (Char *) msg->gppWriteAddr ;          //Recibe la dirección final de los datos enviados
    addr_fisica_DSP = (Char *) msg->imagenDSP_Componentes; //Recibe la dirección inicial de la segunda pool
    size = msg->size;                                //Recibe el tamaño del paquete a procesar
}
```

Fig. 77 Sección de código que adapta los campos del mensaje recibido – núcleo 'C64x+' (función TSKPROCESADO_execute)

Con esa información, se lleva a cabo todo el proceso de descodificación descrito anteriormente (ver apartado 4.3.4), empleando en esta primera fase de reconocimiento de unidades NAL el campo *writebuf*, que apunta al inicio del paquete y que se va incrementando a medida que avanza el procesado del paquete. Una vez procesado éste, en caso de que se hubiera descodificado una imagen, se empleará el campo *addr_fisica_DSP* para copiar la información en la segunda *pool* declarada. Al igual que se hizo previo al envío del mensaje en el 'ARM', ahora se rellenan aquellos campos de la estructura del mensaje en el 'C64x+' necesarios para que posteriormente el 'ARM' sea capaz de interpretar la información de la imagen de forma correcta. Como el puntero al inicio de la segunda *pool* de CMEM ya es conocido por el 'ARM', únicamente se escribe sobre el *flag imagen_disponible*, que valdrá cero o uno en función de si se ha descodificado una imagen o no. Esa operación se realiza en la función `TSKPROCESADO_execute` (ver Fig. 78).

```
//Copia de las dimensiones de la imagen a los campos de la estructura del mensaje
msg->xdimension = XDIM;
msg->ydimension = YDIM;
msg->imagen_disponible=imagenDescodificada;                //Se indica si se ha descodificado imagen o no
status = MSGQ_put (info->gppMsgqQueue, (MSGQ_Msg) msg) ;
```

Fig. 78 Sección de código que rellena la estructura del mensaje – núcleo 'C64x+' (función TSKPROCESADO_execute)

Con el envío del mensaje del 'C64x+' a través de la cola de mensajes (función `MSGQ_put`), éste permanecerá esperando un nuevo mensaje del 'ARM' para procesar el siguiente paquete. La comunicación se da por finalizada hasta el siguiente paquete cuando el 'ARM' recibe ese mensaje y copia la información perteneciente a los campos del mensaje en variables locales para su posterior uso. La Fig. 79 muestra esa recepción y copia de los campos del mensaje llevados a cabo en la función `DSP_Execute`.

```
//Copia de parámetros de la estructura del mensaje a memoria ARM
status = MSGQ_get (SampleGppMsgq, WAIT_FOREVER, (MSGQ_Msg *) &msg) ;
addr_CMEM_DSP_Execute->flag_imagen_descodificada = msg->imagen_disponible;
addr_CMEM_DSP_Execute->xdim_Imagen = msg->xdimension;
addr_CMEM_DSP_Execute->ydim_Imagen = msg->ydimension;
```

Fig. 79 Sección de código que adapta los campos del mensaje recibido – núcleo 'ARM' (función DSP_Execute)

Por último, se presentará un diagrama de flujo (ver Fig. 80) para sintetizar todo el proceso descrito diferenciando las operaciones realizadas en cada uno de los núcleos del procesador OMAP3530. Esa secuencia de operaciones se repetirá tantas veces como paquetes se obtengan del fichero a descodificar.

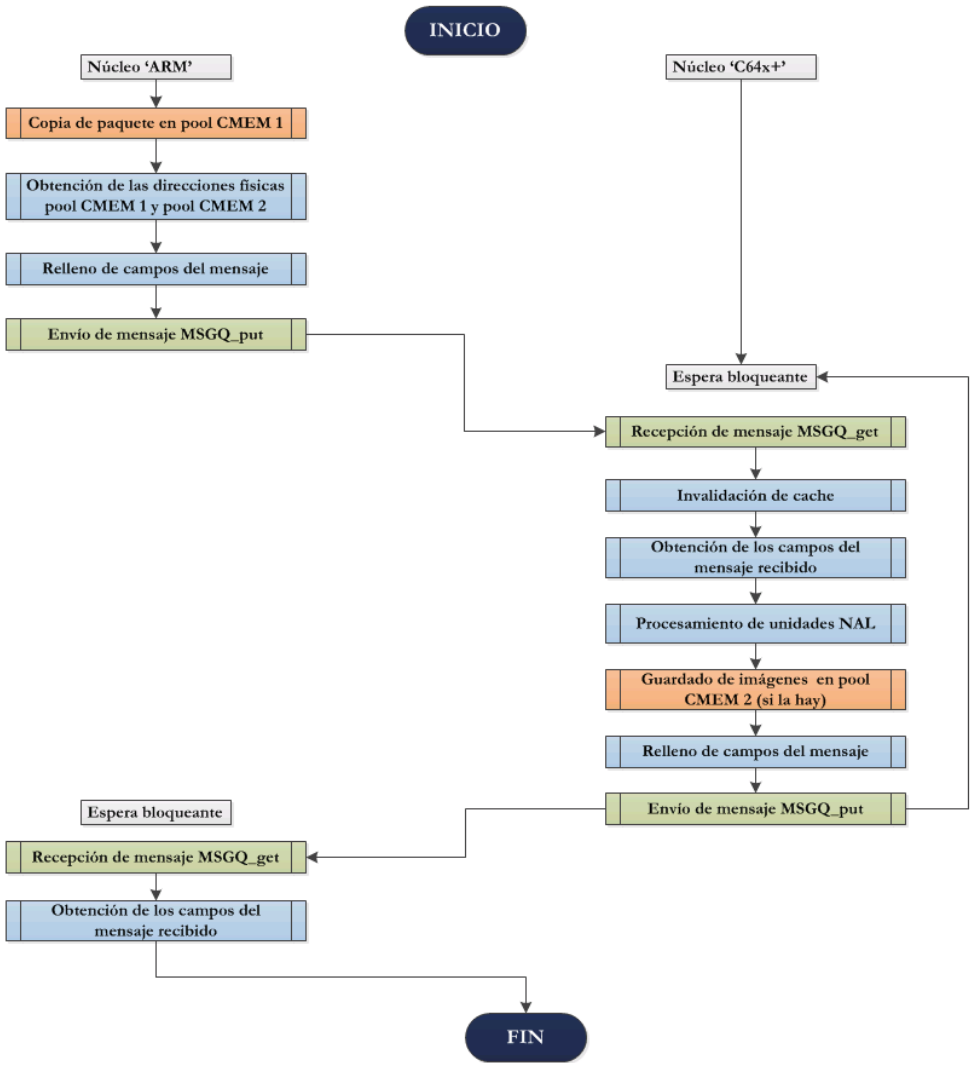

Fig. 80 Diagrama de flujo del proceso de integración

Una vez que el 'ARM' obtiene el mensaje procedente del 'C64x+', el procesador de propósito general comenzará las tareas de presentación de imágenes que serán presentadas a continuación y que constituyen el último apartado del presente capítulo.

4.4 Etapa de salida del descodificador

En este apartado se describirá el proceso de presentación de imágenes, esto es, desde que el reproductor MPlayer obtiene la información de la imagen procedente del descodificador y la graba en variables locales hasta que se visualiza dicha imagen en un monitor. Nuevamente, al igual que sucedió en la etapa de entrada, ha sido necesario analizar detalladamente el flujo de datos del reproductor para comprender cómo se realizan estas tareas. Otra similitud con la primera de las etapas comentadas es el núcleo donde se ejecuta el código de la aplicación encargado de la presentación de imágenes, ya que es realizada exclusivamente por el 'ARM', de modo que no se requiere el uso de los módulos DSPLink y CMEM en esta ocasión.

Situando esta etapa de salida de forma gráfica mediante la Fig. 47 empleada a lo largo del trabajo, la zona destacada de la imagen denota que efectivamente se corresponde con el último de los pasos de la secuencia de operaciones descrita.

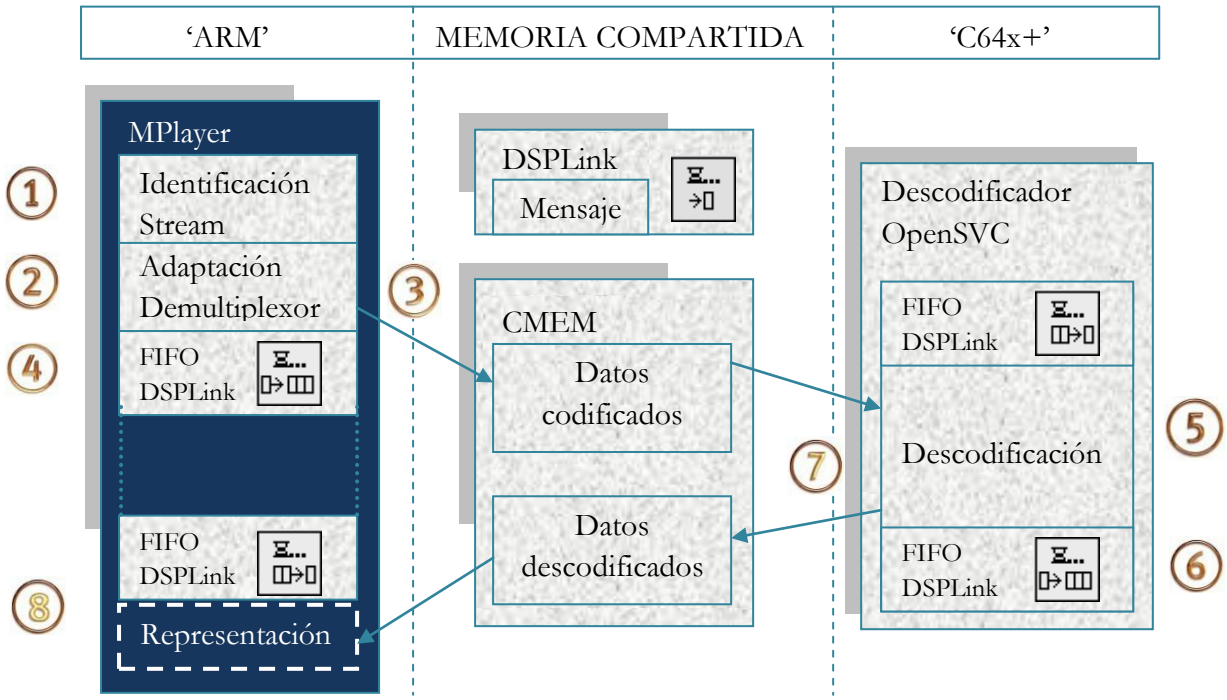

Fig. 47 Diagrama general de la aplicación desarrollada

Para esta etapa, se ha optado por dividir su explicación en los tres subapartados que se presentan a continuación:

- Configuración del controlador de salida de vídeo.
- Adaptación de la información de la imagen, donde se detallarán las modificaciones realizadas para que el 'ARM' disponga de esa información en el lugar adecuado.

- Presentación de imágenes, donde se comentarán aquellas funciones que realizan la escritura de la imagen en el monitor.

Salvo el segundo punto donde sí se han llevado a cabo modifiaciones, en el resto se ha realizado un análisis en profundidad de la información a través de las diferentes funciones que integra el reproductor, ya que desde un primer momento se ha buscado aprovechar al máximo la arquitectura de éste para conseguir el objetivo de este Trabajo Fin de Máster. Al igual que en fases anteriores, la dificultad de ese análisis ha estado en la alta dependencia que presenta MPlayer en cuanto a funciones y paso de estructuras por referencia, dificultando el seguimiento de la información. En consecuencia, el estudio de los subapartados descritos anteriormente se apoyará en diagramas y figuras para lograr una mejor comprensión de los mismos.

4.4.1 Configuración del controlador de salida de vídeo

Como se mencionó en el apartado 2.5, el dispositivo de salida de vídeo empleado va a ser el controlador denominado *Framebuffer* (fbdev en adelante). Desde el punto de vista del *kernel* Linux es un módulo más, de modo que es necesario su carga y configuración, pero al estar soportado por el reproductor multimedia, esas operaciones son realizadas por éste, simplificando el proceso. La elección de este controlador para la representación de contenidos se debe una vez más a mantener la metodología de trabajo de MPlayer. Esto implica que el subsistema DDS del procesador OMAP3530 (ver apartado 2.1.3) emplee el canal *hardware* denominado *graphics* para la gestión de memoria y representación de vídeo. A continuación se describirán dónde se llevan cabo esos procesos, tanto de apertura como de configuración, y algunas particularidades más del controlador de vídeo.

En primer lugar, es importante resaltar que la inicialización del controlador tiene lugar antes de que se descodifique la primera de la imágenes del vídeo escalable. En esta fase inicial únicamente se realiza la apertura del controlador y se comprueba si se ha hecho correctamente. Esta inicialización se realiza en la función `int fb_preinit (int)` ubicada en el fichero `/MPlayer/libvo/vo_fbdev.c`. La Fig. 81 muestra la sección de código donde se realiza esa acción.

```
static int fb_preinit(int reset)
{
    static int fb_preinit_done = 0;     //Flag que indica si se ha inicializado el controlador (estará únicamente a cero tras reset)
    static int fb_works = 0;            //Flag que indica si está activo el controlador (en descodificación estará a 1)

    if (reset) {
        fb_preinit_done = 0;
        return 0;
    }

    if (fb_preinit_done)
        return fb_works;

    fb_dev_fd = fb_tty_fd = -1;

    if (!fb_dev_name && !(fb_dev_name = getenv("FRAMEBUFFER")))     //Nombre del controlador de vídeo empleado
        fb_dev_name = strdup("/dev/fb0");                           //Ubicación del controlador de vídeo
    mp_msg(MSGT_VO, MSGL_V, "using %s\n", fb_dev_name);

    if ((fb_dev_fd = open(fb_dev_name, O_RDWR)) == -1) {            //Apertura del controlador de vídeo
        mp_msg(MSGT_VO, MSGL_ERR, "Can't open %s: %s\n", fb_dev_name, strerror(errno));
        goto err_out;
    }
```

Fig. 81 Sección de código que inicializa el controlador de salida de vídeo (función fb_preinit)

En la imagen se presenta la apertura del controlador de vídeo seleccionado previamente así como los *flags* empleados para determinar el estado del controlador: el primero de ellos, *fb_preinit_done*, presentará un nivel lógico '0' antes de realizar la configuración inicial y, una vez hecha, permanecerá a '1' hasta el próximo arranque de la aplicación, mientras que el segundo, *fb_works*, valdrá '1' durante el funcionamiento normal del controlador, una vez inicializado éste.

A continuación, se lleva a cabo la descodificación de la primera imagen, de tal forma que la información obtenida de ésta (dimensiones, número de *bits* por pixel, formato de muestreo, espacio de color, etc.), será empleada para la configuración del controlador. En consecuencia, las características de la imagen determinarán los parámetros de configuración del controlador. Para el caso que se presenta, las dimensiones de la imagen dependerán de la resolución espacial del vídeo a descodificar pero en cuanto al formato de muestreo y espacio de color, el descodificador OpenSVC proporciona las imágenes en formato de muestreo 4:2:0[27] y el espacio de color es YUV independientemente del vídeo elegido. Estos dos últimos parámetros son detectados por MPlayer en la fase previa a la configuración del controlador, por lo que la función encargada de esta acción, `int mpcodecs_config_vo (sh_video_t *, int , int, unsigned int)` ubicada en el fichero `/MPlayer/libmpcodecs/vd.c`, recibirá como parámetros de entrada las dimensiones de la imagen y el espacio de color detectado por el reproductor. La Fig. 82 muestra la sección de código donde se realiza la configuración del controlador así como el resultado a través del terminal de consola.

```
int mpcodecs_config_vo(sh_video_t *sh, int w, int h, unsigned int preferred_outfmt)
{
     .........
// Time to config libvo!
mp_msg(MSGT_CPLAYER, MSGL_V,
       "VO Config (%dx%d->%dx%d,flags=%d,'%s',0x%X)\n", sh->disp_w,
       sh->disp_h, screen_size_x, screen_size_y, vocfg_flags, "MPlayer", out_fmt);
```

Visualización en el terminal de consola (linea serie)

```
VDec: using Planar YV12 as output csp (no 0)
Movie-Aspect is undefined - no prescaling applied.
VO Config (176x144->176x144,flags=0,'MPlayer',0x32315659)
```

Dimensiones de la imagen

Ajuste de dimensiones en el monitor de presentación

Se corresponde con el formato detectado previamente por MPlayer: YV12

Fig. 82 Sección de código que realiza la configuración inicial del controlador de salida de vídeo (función mpcodecs_config_vo)

[27] En el formato de muestreo 4:2:0, las dos crominancias se submuestrean de forma que su resolución es una cuarta parte respecto de la luminancia. Cada crominancia representa la información de color correspondiente a la posición de las cuatro luminancias adyacentes, realizando un submuestreo tanto horizontal como vertical. En cuanto al orden, primero va la luminancia íntegramente y a continuación las dos crominancias sin mezclarse, esto es, primero una y luego otra.

La función, mediante los parámetros de entrada recibidos, se encarga de configurar el controlador de vídeo[28], de tal forma que en principio ya se podrían presentar las imágenes descodificadas en el monitor. Sin embargo, este controlador únicamente soporta los siguientes espacios de color: RGB565, RGB444 y RGB888, por lo que es necesario realizar algún tipo de conversión para poder transformar la imagen en formato YUV 4:2:0 a uno de los formatos soportados por el controlador. El reproductor MPlayer es capaz de detectar esta problemática y, en concreto, proporcionar una conversión *software* del formato YUV4:2:0 al formato RGB565 [39].

Este nuevo espacio de color, RGB565, emplea 16 *bits* para codificar la cada pixel de la imagen, dando mayor número de *bits* al color verde, seis, por los cinco empleados tanto para el color azul como para el rojo. Esto se debe a la mayor sensibilidad del ojo respecto al primer color (verde) frente a los dos últimos.

Por último, MPlayer imprime por la terminal (ver apartado 4.1) información perteneciente a la dirección de memoria donde se copiarán los datos que se quieren representar en el monitor además del nuevo espacio de color configurado (RGB565) e información perteneciente a lo que MPlayer denomina escalado, esto es, la conversión del espacio de color YUV a RGB565 (ver Fig. 83).

```
SwScale: scaling 176x144 Planar YV12 to 176x144 BGR 16-bit
[swscaler @ 0x940b80]No accelerated colorspace conversion found from yuv420p to rgb565le.
[swscaler @ 0x940b80]using unscaled yuv420p -> rgb565le special converter
REQ: flags=0x403  req=0x0
VO: [fbdev] 176x144 => 176x144 BGR 16-bit
VO: Description: Framebuffer Device
VO: Author: Szabolcs Berecz <szabi@inf.elte.hu>
Can't set graphics mode: Invalid argument
framebuffer too small for double-buffering, disabling
frame_buffer @ 0x43064000
center @ 0x43064000
pixel per line: 800
```

Fig. 83 Escalado *software* y configuración definitiva del controlador de salida de vídeo

Con esta información, el controlador fbdev queda configurado para su posterior empleo en la presentación de imágenes descodificadas en el monitor. Anotar que este proceso descrito se realiza una sola vez, mientras que el escalado *software* para pasar de un espacio de color a otro se realiza una vez por cada imagen representada en el monitor. La función encargada de la conversión no ha sido analizada, ya que este paso se considera únicamente funcional, pero sí que ha sido localizada dentro de la estructura de directorios de MPlayer por si fuera necesario modificarla. La llamada a la función se realiza a través de una macro, denominada `YUV2RGBFUNC`, ubicada en el fichero `/MPlayer/libswscale/yuv2rgb.c`.

4.4.2 Adaptación de la información de la imagen

Tal y como se indicó en la introducción de este apartado, ha sido necesario realizar una serie de modificaciones (ver apartado 4.3.3) en algunas estructuras pertenecientes a MPlayer para transferir la información de la imagen descodificada por OpenSVC al lugar adecuado.

[28] En la Fig. 86 se indica que el espacio de color detectado por MPlayer es YV12 y no YUV420 pero ambos espacios son interpretados exactamente igual por el reproductor. Para más información acudir al archivo REFcolorspaces.txt ubicado en /Mplayer/DOCS/tech/.

En un principio, se estudio la opción de modificar los prototipos de funciones propias del reproductor, añadiendo un parámetro más de entrada a la propia función. Sin embargo, el alto grado de dependencia que presenta MPlayer por su extenso número de ficheros y funciones hizo inviable esta opción. Como alternativa se pensó en la posibilidad de incorporar los campos necesarios de la imagen descodificada en aquellas estructuras que estuvieran involucradas en esta tarea, algo que requeriría menos cambios. Esta segunda opción ha sido la elegida finalmente,y para ello se ha realizado nuevamente un análisis con el fin de conocer cuáles son las estructuras que transfieren la información de la imagen descodificada.

Dado que no es objetivo de este proyecto implementar un nuevo método de trabajo para esta tarea, se ha buscado nuevamente aprovechar la arquitectura que proporciona el reproductor para la presentación de imágenes. Para ello, por un lado se ha hecho uso de la estructura creada en el fichero de encabezado *libreriaOHA.h* (ver apartado 4.3.3), incorporando un campo de tipo `infoCMEM_t_imagen` en la estructura `demux_stream.h` ubicada en el fichero `/MPlayer/libmpdemux/demuxer.h`, mientras que por otro lado se han añadido una serie de campos a las estructuras denominadas `sh_video_t` y `AVCodecContext` ubicadas en los ficheros `/MPlayer/libmpdemux/stheader.h` y `/MPlayer/libavcodec/avcodec.h` respectivamente. Esos campos se corresponden con la dirección de memoria (inicio de la 2ª *pool* CMEM) donde comienzan los datos pertenecientes a la imagen descodificada (`imagen_datos_nombre_estructura`), las dimensiones de la imagen (`dimensión_x` y `dimensión_y`) y un indicador para conocer si se ha descodificado una imagen o no (`flag_imagen_descodificadaDSP`).

En la Fig. 84 se ofrecen las modificaciones realizadas en las estructuras mencionadas, mostrándolas en el orden seguido en la explicación.

Fichero demuxer.h

```
typedef struct {
  .........
  //Añadido aquí ya que es el lugar idóneo para los casos estudiados. Se evita modificar prototipos de funciones genéricas
  infoCMEM_t_imagen addr_CMEM;  //Sin puntero!

}demux_stream_t;
```

Fichero stheader.h

```
typedef struct sh_video {
    .........
    unsigned int * imagen_datos_sh_video_t;
    int dimension_x;
    int dimension_y;
    int flag_imagen_descodificadaDSP;
} sh_video_t;
```

Fichero avcodec.h

```
typedef struct AVCodecContext {
    .........
    //Parámetros donde se copiará la info de la imagen descodificadaDSP.
    unsigned int * imagen_datos_AVCodecContext;
    int dimension_x_AVCodecContext;
    int dimension_y_AVCodecContext;
    int flag_imagenDSP_AVCodecContext;
} AVCodecContext;
```

Fig. 84 Adición de campos de la imagen en el reproductor MPlayer

Para describir la transferencia de información entre estas estructuras se ha optado por realizar un diagrama de flujo (ver Fig. 85) representado el proceso completo, es decir, desde que se recibe el mensaje procedente del núcleo 'C64x+' con la información de la imagen descodificada hasta que se copia la información de dicha imagen en el lugar adecuado para

su posterior presentación en el monitor a través del controlador de vídeo. Cada subproceso del diagrama lleva asociada las operaciones realizadas en él así como su ubicación en la estructura de ficheros de MPlayer.

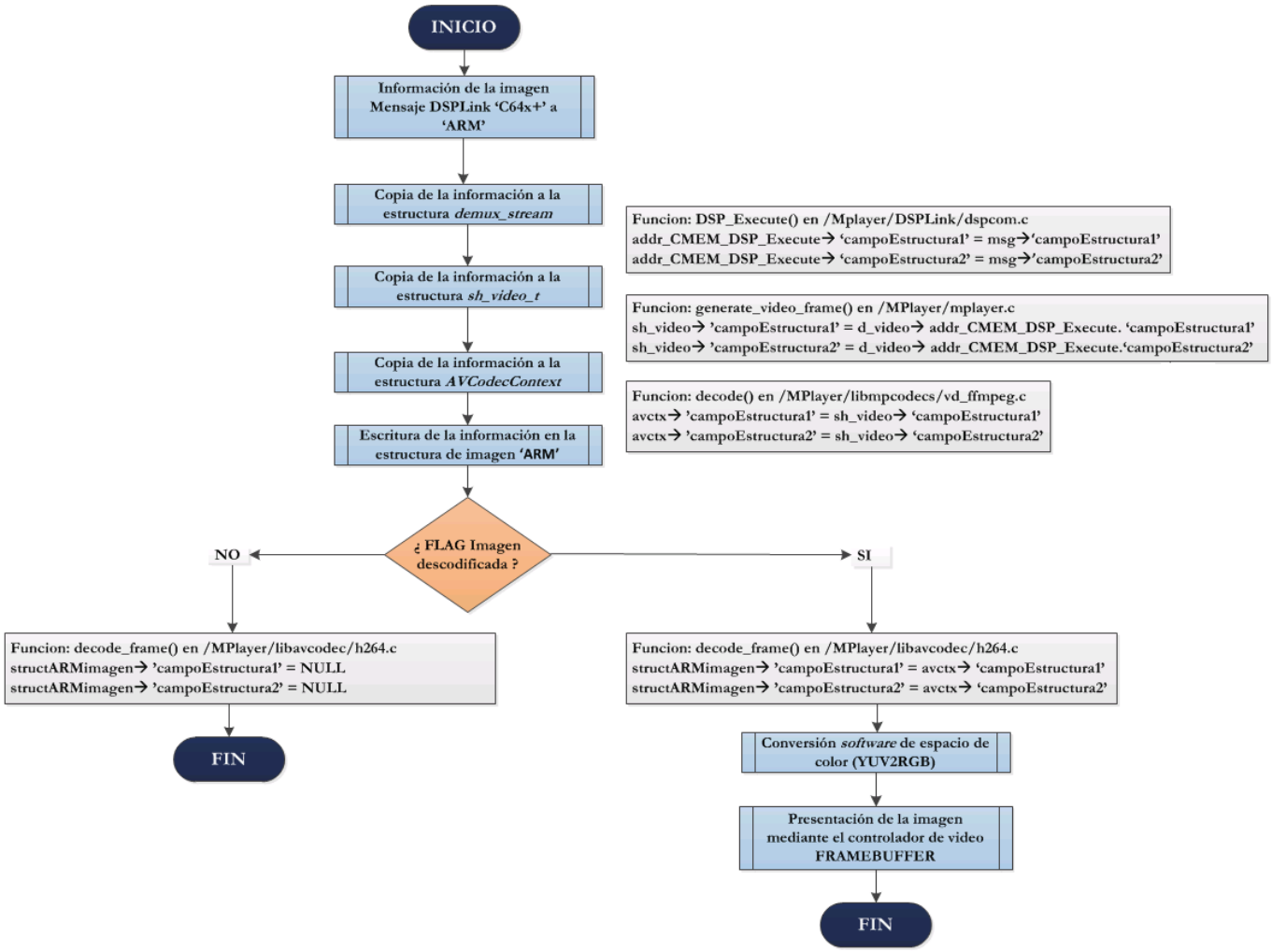

Fig. 85 Transferencia de información entre las estructuras modificadas

Cada uno de los campos mencionados en la Fig. 84 es copiado en su campo dentro de la estructura que corresponda en cada momento. La única diferencia en cuanto a comportamiento se obtiene a través del *flag imagen_descodificada,* cuyo valor indicará si OpenSVC ha descodificado una imagen o no.

Si el resultado es positivo (ver Fig. 86):

- Se emplea el puntero que contiene la dirección de memoria del inicio de la segunda *pool* de CMEM y se opera con él para que el núcleo 'ARM' conozca el inicio (la dirección de memoria) de cada componente de la imagen (luminancia y las dos crominancias), almacenando esa información en los campos `data[0]`, `data[1]` y `data[2]` de la estructura de tipo `AVFrame`. Es importante resaltar que se opera directamente desde la segunda *pool* de CMEM, de forma que se evita una nueva copia local de datos en memoria del núcleo 'ARM'.

Si el resultado es negativo:

- Se escribe NULL (vacío) en cada una de las componentes de la imagen y no se ejecuta la sección perteneciente a la conversión y posterior presentación de la imagen.

```
pict -> data[0] = (unsigned char *)avctx->imagen_datos_AVCodecContext;          //Luminancia
pict -> data[1] = (unsigned char *)(pict -> data[0] +                           //1ªcrominancia
                  (avctx->dimension_x_AVCodecContext+32)*avctx->dimension_y_AVCodecContext);
pict -> data[2] = (unsigned char *)(pict -> data[1] +                           //2ªcrominancia
                  ((avctx->dimension_x_AVCodecContext+32)*(avctx->dimension_y_AVCodecContext)/4));
```

Fig. 86 Sección de código que obtiene los punteros a cada componente de la imagen descodificada (función decode_frame)

Conocidas las adaptaciones necesarias para que el reproductor MPlayer pueda recibir correctamente la información perteneciente a cada una de las imágenes descodificadas por OpenSVC, el siguiente y último apartado describirá de forma breve las funciones encargadas de realizar la representación de la imagen en el monitor.

4.4.3 Presentación de imágenes

Una vez realizada la conversión de cada imagen del espacio de color YUV a RGB565, las funciones encargadas de la representación de la información en el monitor se denominan `int draw_slice (uint8_t *, int, int, int, int, int)` y `void memcpy_pic2 (void *, const void *, int, int, int, int, int)` ubicadas en los ficheros `/MPlayer/libvo/vo_fbdev` y `/MPlayer/libvo/fastmemcpy.h`. Al igual que sucedió en el caso anterior, únicamente ha sido necesario analizar los parámetros de entrada/salida que recibe la función para asegurar que los contenidos presentados son los descodificados por OpenSVC. Ya que la segunda de las dos funciones presentadas es simplemente una copia de información del contenido de la imagen al controlador de salida de vídeo, se ha preferido detallar cada uno de los campos que recibe esta función desde la función llamante, `draw_slice`, para una mejor comprensión de la tarea realizada (ver Fig. 87).

```
static int draw_slice(uint8_t *src[], int stride[], int w, int h, int x, int y)
{
    uint8_t *d;
    //center se corresponde con el puntero obtenido en la configuración del controlador
    //'x' e 'y' permiten seleccionar la zona de "pintado" de la imagen. Para este TFM valen 0
    d = center + fb_line_len * y + fb_pixel_size * x;
    /*
    Parámetros de la función
    d: dirección de memoria que apunta al propio "monitor", es decir, al primer pixel de la imagen a representar en el monitor
    src[0]: dirección de memoria que apunta a la información de la imagen descodificada (ya convertida a RGB565)
    stride[0]: ancho de la línea horizontal (resolución horizontal de la imagen)
    w*h son las dimensiones de la imagen descodificada
    fb_pixel_size: parámetro de configuración del controlador FRAMEBUFFER y vale 2(tamaño pixel) en unidades de byte
    fb_line_len:  parámetro de configuración del controlador FRAMEBUFFER y vale 800 (resolución horizontal del monitor)
    */
    memcpy_pic2(d, src[0], w * fb_pixel_size, h, fb_line_len, stride[0], 1);

    return 0;
}
```

Fig. 87 Sección de código que realiza la representación de la imagen descodificada en el monitor (función draw_slice)

Con esta última fase da por terminado el presente capítulo en el que se han estudiado y analizado las tres etapas de la aplicación construida: previa al descodificador, la integración del reproductor con el descodificador y posterior al descodificador. En cada una de ellas se ha tratado de ofrecer una versión práctica de las operaciones realizadas, con figuras y diagramas de flujo que sirvan para aclarar los contenidos descritos. El capítulo cinco presentará las pruebas realizadas a lo largo del trabajo desarrollado para integrar los diferentes elementos del sistema (módulos, correcto envío y recepción de información, etc.) y analizará el rendimiento del descodificador con diferentes secuencias escalables de entrada. Antes de finalizar, se ofrece un diagrama de flujo (ver Fig. 88) en el que se describen de forma muy sintetizada las fases principales de cada etapa y el núcleo del procesador OMAP3530 en el que tienen lugar.

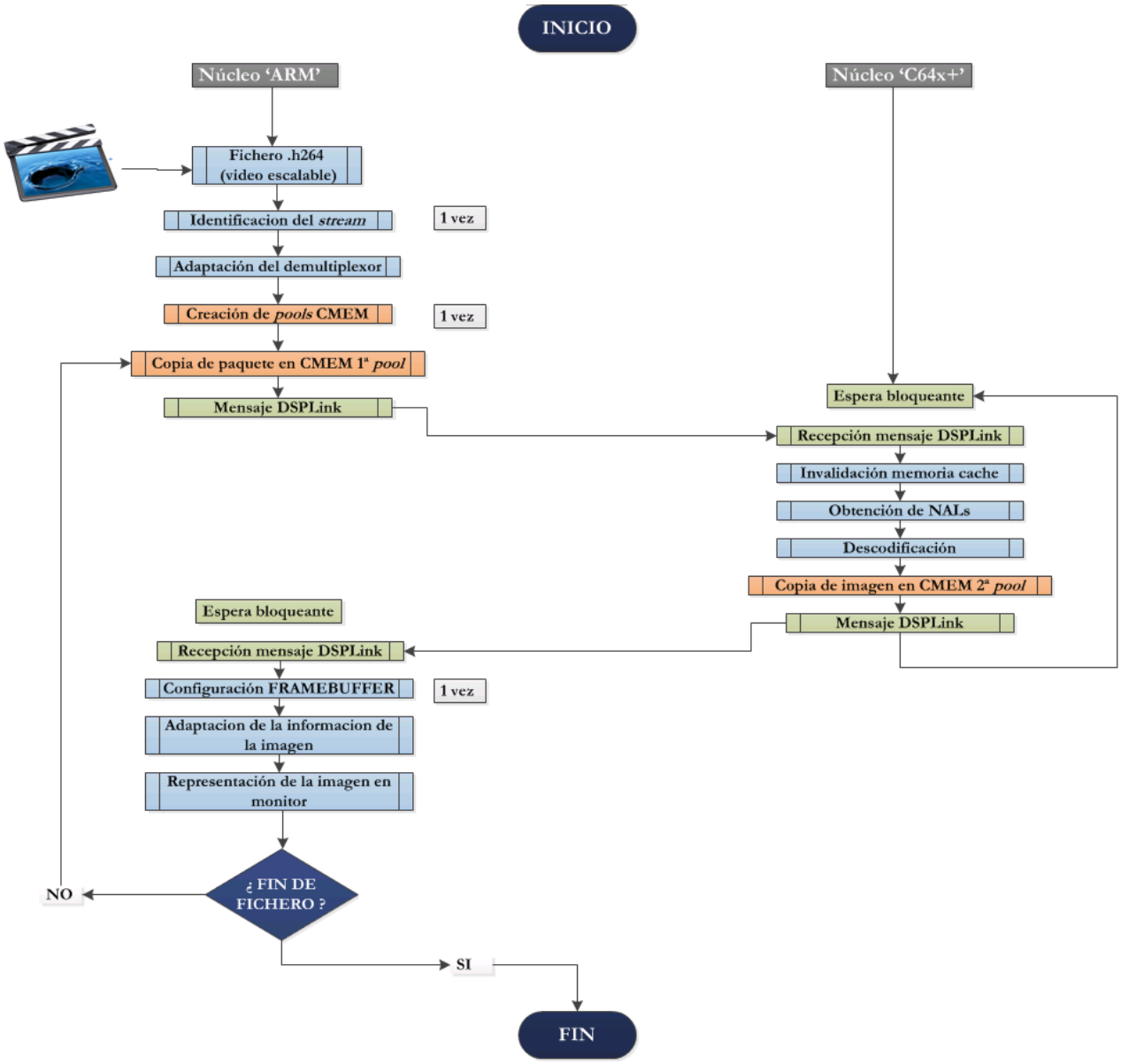

Fig. 88 Diagrama de flujo sintetizado de la aplicación desarrollada

5 INTEGRACIÓN DEL SISTEMA Y RESULTADOS

En este capítulo se presentan las diferentes pruebas parciales realizadas a lo largo del desarrollo de este Trabajo Fin de Máster así como una serie de medidas de rendimiento del descodificador OpenSVC.

En primer lugar, se describe cada una de las fases en las que se ha ido realizando la integración, las cuales han detectado problemas importantes cuya solución ha supuesto una gran dedicación y esfuerzo.

En segundo lugar, se presentan los resultados obtenidos asociados al rendimiento del descodificador OpenSVC empleando una serie de secuencias de pruebas que serán descritas previamente.

Asociado a las medidas de rendimiento del descodificador y a trabajos previos realizados en el Grupo de Investigación GDEM, se establecerá una comparativa entre diferentes arquitecturas empleadas en la descodificación de un vídeo escalable, extrayendo una serie de conclusiones que finalizarán el presente capítulo.

5.1 Introducción

La integración de MPlayer y OpenSVC ha generado numerosos problemas en prácticamente todas las etapas que se han descrito en el capítulo anterior. Al estar compuesta la aplicación por diferentes elementos, todos ellos de elevada complejidad, han ido apareciendo problemas de menor/mayor dificultad que ha sido necesario resolver para realizar la integración correctamente.

Como se ha seguido un orden secuencial en la integración del reproductor y el descodificador, se han ido realizando pruebas parciales en el desarrollo del trabajo para garantizar el correcto funcionamiento de cada tarea. De este modo, se busca acotar el problema de forma sencilla, localizando al menos la fuente de error rápidamente. Mediante este procedimiento se evitan retrocesos de mayor magnitud en el desarrollo del trabajo, ganando en eficiencia y evitando repetir procesos muy iniciales. Además, el funcionamiento de la aplicación ha permitido realizar la caracterización del descodificador OpenSVC a través de una serie de medidas de rendimiento con un conjunto de secuencias que serán descritas posteriormente.

El presente capítulo se encuentra dividido en dos partes: la primera de ellas describirá una serie de pruebas parciales, ordenadas en base a la realización de este trabajo, esto es, comenzando por tareas iniciales del desarrollo y terminando por la representación de imágenes en el monitor de salida. Como algunas de ellas han generado muchos problemas, se ha optado por definir una serie de pasos intermedios necesarios para realizar la tarea correspondiente. Por otro lado, la segunda parte del capítulo está dedicada a la medida del rendimiento del descodificador. En primer lugar, se realizará una descripción de las secuencias empleadas así como del procedimiento de medida; posteriormente se presentarán los resultados del descodificador OpenSVC y se hará una comparativa con los resultados de trabajos previos realizados en el GDEM[29]; por último se extraerán una serie de conclusiones asociados a esas comparativas.

5.2 Fases parciales de la integración

En este apartado se describirán las diferentes pruebas parciales realizadas a lo largo del desarrollo del trabajo. Para algunas de ellas se definirán un conjunto de pasos a seguir para llevar a cabo las tareas en el orden adecuado mientras que otras pruebas detectaron fuentes de error que también serán comentados a continuación.

La Fig. 47 (ver apartado 4.1), empleada a lo largo del capítulo anterior con el fin de enlazar y ubicar los diferentes elementos que forman la aplicación desarrollada en este Trabajo Fin de Máster, será la referencia nuevamente para los apartados que se presentan a continuación. En concreto, los tres primeros apartados (5.2.1, 5.2.2 y 5.2.3) son etapas más iniciales en el desarrollo, mientras que los apartados restantes se ubican en pleno proceso de integración de MPlayer y OpenSVC.

[29] En el GDEM se han realizado medidas de rendimiento con el mismo sistema embebido pero empleando una versión aislada y optimizada del descodificador OpenSVC en el núcleo 'C64x+'.

5.2.1 Configuración del mapa de memoria de los núcleos 'ARM' y 'C64x+'

El mapa de memoria de los núcleos del procesador OMAP3530 debe ser exactamente igual para poder ejecutar una aplicación en la que ambos núcleos interactúen sin errores. De este modo, tal y como se indicó en el apartado 3.2.2.1.4, es de significativa importancia fijar una estructura de memoria, con sus secciones claramente identificadas en sus respectivos rangos y, a partir de ahí, seguir con el desarrollo de las aplicaciones. Además, el modo en el que se realizan cambios en el mapa de memoria difiere notablemente en función del núcleo ('ARM' o 'C64x+').

En el caso de este trabajo, la configuración de la memoria está directamente relacionada con los módulos DSPLink y CMEM, más concretamente con la obtención de cada uno de los módulos en sí misma (ficheros con extensión **.ko*). Por esta razón, es importante diferenciar entre los cambios realizados antes de la generación del módulo y después de la generación de éste. Si esta distinción se analiza para cada núcleo del procesador OMAP3530, en el caso del 'C64x+' es independiente el momento de obtención del módulo de la configuración de la memoria del núcleo, ya que ésta se realiza por medio de ficheros de configuración del sistema operativo DSP/BIOS (extensión **.tcf*). Sin embargo, para el 'ARM', el orden en el cual se realicen las operaciones sí que es dependiente de la obtención de los módulos, ya que el proceso de configuración de la memoria del 'ARM' se realiza a través de la configuración del módulo DSPLink.

A continuación se define el conjunto de pasos necesarios para configurar correctamente el mapa de memoria del núcleo 'ARM' a través del proceso de configuración de DSPLink.

1. Configuración de las secciones de memoria (tamaño, rangos de direcciones, nombre, etc.) a través del fichero `CFG_OMAP3530_SHMEM.c` ubicado en el directorio `(DSPLINK)/config/all/`.
2. Regeneración del módulo *dsplink.ko* (ver apartados desde 3.2.2.1.1 al 3.2.2.1.5).
3. Sustitución del módulo anterior por el recién obtenido en el directorio específico del sistema de ficheros Angström ubicado en el directorio `/lib/modules/2.6.32/kernel/drivers/dsp` (ver apartado 3.2.2.3).
4. Actualización de las librerías tanto para 'ARM' como para 'C64x+' en el reproductor MPlayer, con el fin de emplear la nueva configuración realizada.
5. Repetición del proceso de compilación y obtención del ejecutable para las aplicaciones de 'ARM' y 'C64x+'.

No se ha especificado en ninguno de los pasos la configuración del módulo CMEM, pero tal y como se indicó en el apartado 3.2.2.1.4, el tamaño y los rangos de direcciones de memoria establecidos deben de estar incluidos en el fichero `CFG_OMAP3530_SHMEM.c`. Una vez definidos esos parámetros, la obtención del módulo CMEM es independiente del proceso de configuración descrito.

5.2.2 Inclusión de las librerías en el *makefile* de MPlayer

Un paso imprescindible en la configuración previa del reproductor antes de realizar modificaciones en él es la adición de las librerías pertenecientes a los módulos DSPLink y

CMEM en el lugar adecuado. Esta operación permitirá emplear las APIs de ambos módulos en el código de MPlayer (ver apartado 4.3.3).

La generación de esas librerías se realiza en el mismo proceso de obtención de los módulos *dsplink.ko* y *cmemk.ko*. En consecuencia, será necesario en primer lugar modificar los *makefiles* de los módulos DSPLink y CMEM, indicando la ubicación de las fuentes del *kernel* a emplear, cuya versión deberá coincidir con el *kernel* Linux del sistema embebido. A continuación, se hará uso nuevamente de la utilidad *make* para obtener tanto el módulo como las librerías pertenecientes a cada módulo. Por último, esas librerías serán añadidas al *makefile* pero ahora del reproductor, localizando previamente la ubicación correcta dentro del fichero. Al ser el *makefile* de MPlayer un fichero complejo, con uso masivo de macros, es necesario identificar claramente el lugar donde serán incluidos los directorios pertenecientes a las librerías generadas para su posterior uso en el código del reproductor. Es importante destacar que el conjunto de operaciones descrito se realiza exclusivamente para el núcleo 'ARM'.

Para finalizar, se muestran de forma sintetizada las operaciones descritas de forma secuencial:

1. Modificación de los *makefiles* de los módulos DSPLink y CMEM indicando en ellos la versión del *kernel* para la cual son compilados, siendo necesario que sea la misma que la del *kernel* Linux del sistema embebido.
2. Obtención de las librerías *dsplink.a* y *cmemd.a470uC* tras previa ejecución de la utilidad *make*.
3. Inclusión de las librerías obtenidas en el *makefile* de MPlayer (ver apartado 4.3.3).

5.2.3 Conexión con el 'C64x+' mediante DSPLink

La conexión con el núcleo 'C64x+' se ha simplificado considerablemente respecto a versiones anteriores de CCS dadas las mejoras introducidas en cuanto a depuración multiprocesador la versión 5.1 del entorno de desarrollo integrado (ver apartado 3.3.3). Pese a ello, la novedad de esa característica ha provocado que la documentación asociada a ese tema sea escasa, haciendo que el procedimiento para depurar aplicaciones simultáneamente en ambos núcleos fuera costoso y complejo.

Como se ha comentado en varias ocasiones a lo largo del documento, el núcleo 'ARM' tiene control total sobre el 'C64x+' (jerarquía *master-slave*), de forma que el enlace, la carga del programa y la inicialización del 'C64x+' son tareas realizadas por el 'ARM'. Las funciones que implementan esas tareas pertenecen a la API de DSPLink, por lo que nuevamente el módulo es empleado en tareas de especial importancia.

También es importante destacar que el emulador conectado mediante la interfaz JTAG al sistema embebido BeagleBoard proporciona información acerca del estado de operación del núcleo 'C64x+' en modo depuración: sin conectar, en estado de *reset* o conectado. Recordando la información del apartado 3.3.3, en concreto la Fig. 40, el 'C64x+' permanecerá en estado de *reset* hasta la ejecución de una serie de funciones (`PROC_Setup`,

`PROC_Attach`, `PROC_load` y `PROC_Start`) que permitirán la conexión con el procesador digital de señal.

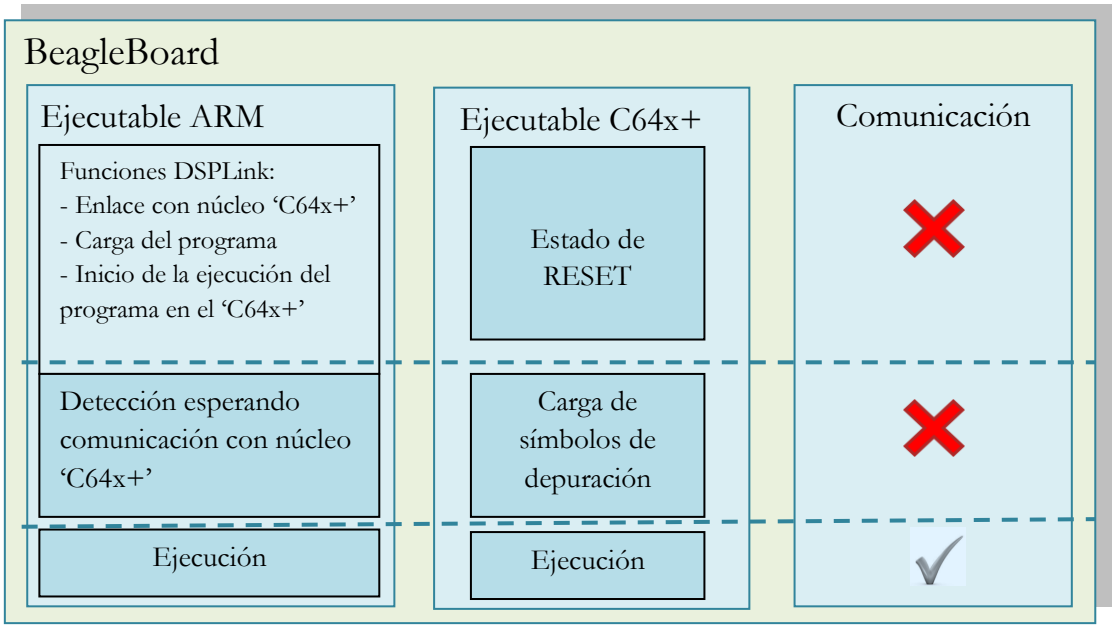

Fig. 40 Procedimiento de conexión entre los núcleos del procesador OMAP3530

Asociado a la figura, los mensajes del emulador que indican el estado de operación del núcleo se muestran a continuación:

- Conexión no establecida con el emulador. Posible desconexión física de sus interfaces (JTAG al sistema emebido o USB al PC de desarrollo).

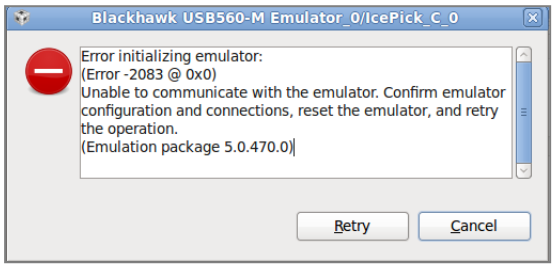

Fig. 89 Conexión no establecida con el emulador

- Conexión no establecida con el emulador. Dispositivo en estado de *reset*.

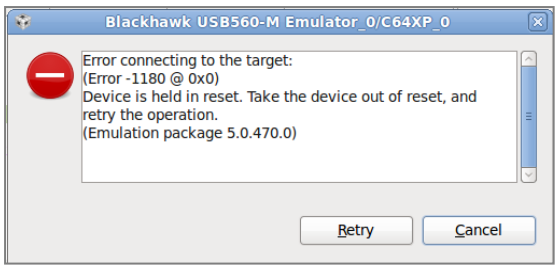

Fig. 90 Núcleo 'C64x+' en estado de *reset*

Ya que el procedimiento para la conexión del núcleo 'C64x+' en modo depuración fue descrito en el apartado 3.3.3, se comentará a continuación el método de conexión empleado para la ejecución de la aplicación en una terminal de consola, sin posibilidad de colocación de puntos de ruptura y otros elementos de depuración. Aunque parezca un tema sin aparente complejidad, la solución obtenida no se ha logrado hasta las últimas etapas del desarrollo de este Trabajo Fin de Máster, siendo necesario recurrir siempre al modo depuración, con el hándicap en cuanto a tiempo que esto supone.

El principal problema asociado al lanzamiento de la aplicación en una terminal de consola es la falta de sincronización entre los núcleos del procesador OMAP3530 en las tareas iniciales. Como se ha visto en el modo depuración, es necesario parar el código del núcleo 'ARM' en un momento determinado para liberar al 'C64x+' del estado del *reset* y que comience la ejecución de su código, que se corresponde con la inicialización del descodificador OpenSVC. Para ello, al no disponer de puntos de ruptura en este modo, se ha procedido a colocar las siguientes instrucciones (ver Fig. 91) a continuación de la función `MSGQ_transportOpen`, localizada a su vez en la función `DSP_Create(IN Char8 *,IN Char8 *,IN Uint8)`.

```
//Open the remote transport.
if (DSP SUCCEEDED (status)) {
    status = MSGQ_transportOpen (processorId, &mqtAttrs) ;

    if (DSP_FAILED (status)) {
        RDWR_1Print ("MSGQ_transportOpen () failed. Status: [0x%x]\n",
                     status) ;
    }
    // Add the two lines below OHA 25-6-12 para la ejecución de la app en línea de comandos
        printf ("Hit Enter to continue(MSGQ_TransportOpen)...\n");
        getchar();
}
```

Fig. 91 Sección de código necesaria para la ejecución de aplicaciones desde la terminal (función DSP_Create)

La función *getchar* se encarga de recoger un carácter introducido por el teclado, realizando una espera bloqueante que permite al núcleo 'C64x+' salir del estado del *reset* e iniciar su ejecución. La referencia consultada para realizar esta operación[40] no se ha materializado por completo en el código ya que indicaba la colocación de la función *getchar* después de la ejecución de la función `PROC_Start`. Esta opción se probó y no resultó, generando un error el 'ARM' de escritura en una zona de memoria no destinada a ello. Por tanto, se pensó en la posibilidad de colocar dicha función bloqueante después de la apertura del canal de comunicación que permite la transferencia de mensajes entre núcleos. De esta manera, el error de memoria desaparece y se logra la comunicación de los dos núcleos del procesador OMAP3530 exitosamente.

5.2.4 Copia de datos correcta en las *pools* de CMEM

Para la verificación de contenidos copiados en las dos *pools* declaradas por el módulo CMEM, se han seguido procedimientos diferentes para los núcleos del procesador OMAP3530 debido a la problématica de las direcciones virtuales manejadas por núcleo 'ARM'. La primera opción para visualizar el contenido de la memoria compartida fue emplear la utilidad *Memory Browser* de CCS (ver Fig. 92), representando su contenido en código hexadecimal. Esta utilidad muestra el contenido de cada posición de memoria para cada núcleo, de forma que un buen método para comprobar si, por ejemplo, los paquetes codificados y copiados en la primera *pool* de CMEM por el núcleo 'ARM' se corresponden con los leídos en el 'C64x+' es visualizar dicha utilidad. El objetivo de ver los datos de cada posición de memoria no es comparar *byte* por *byte* su valor, sino tener al menos una referencia genérica de que la memoria compartida está funcionando como debería.

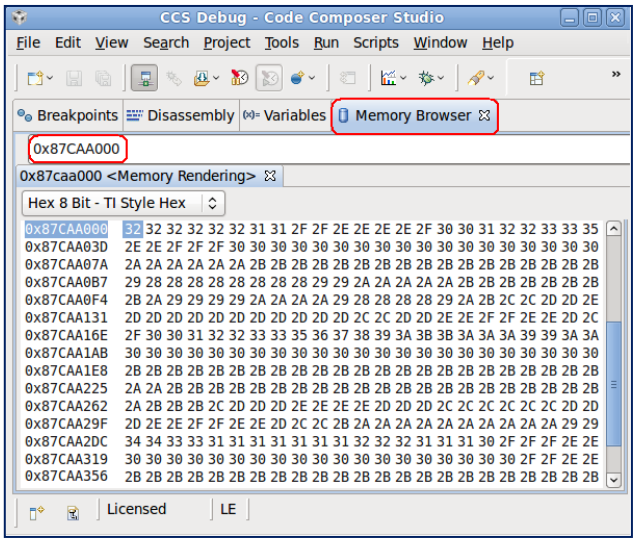

Fig. 92 Utilidad *Memory Browser* de CCS

Sin embargo, aparece el primer problema asociado a las ya mencionadas direcciones virtuales. CCS no es capaz de visualizar ninguna dirección de memoria virtual que pertenezca al rango declarado por CMEM. Esto significa que no es posible comparar si los datos copiados en un núcleo son los mismos que los leídos por el otro y viceversa (para todas las *pools* declaradas por el módulo). Por tanto, es necesario buscar otras alternativas para el núcleo 'ARM'.

En concreto, se ha empleado la función `memcmp`, la cual recibe las direcciones iniciales (punteros) origen y destino pertenecientes a las dos fuentes de datos a comparar. El tercer parámetro de la función es el número de *bytes* a comparar. Anterior a la comparación se realiza la copia de datos en memoria compartida y la ejecución de la función `memcmp` devuelve uno de los siguientes resultados:

- Mayor que 0, si el objeto apuntado por la dirección origen es mayor que el objeto apuntado por la dirección destino.
- Menor que 0, si el objeto apuntado por la dirección origen es menor que el objeto apuntado por la dirección destino.
- 0, si son iguales.

La Fig. 93 muestra el código necesario para emplear la función mencionada con los parámetros adecuados:

- `dp->buffer`: puntero al inicio del paquete de datos codificado
- `addr_CMRM.addr_poolcmem1`: puntero al inicio de la primera *pool* de CMEM
- `pkt.size`: tamaño total del paquete de datos codificado en *bytes*.

```
//Se comprueba que los buffers son iguales ... Quitar comprobación cuando se este seguro
char n=memcmp ( dp->buffer, addr_CMEM.addr_poolcmem1, pkt.size );
  if (n>0) printf ("'%s' is GREATER than '%s'.\n",dp->buffer,addr_CMEM.addr_poolcmem1);
    else if (n<0) printf ("'%s' is LESS than '%s'.\n",dp->buffer,addr_CMEM.addr_poolcmem1);
        else printf ("'%s' is the SAME as '%s'.\n",dp->buffer,addr_CMEM.addr_poolcmem1);
```

Fig. 93 Sección de código que realiza la comparación de datos para la primera *pool* de CMEM

Una vez comprobadas que las copias de datos se hacen correctamente en la memoria compartida, es preferible eliminar esta sección de código para acelerar la ejecución de la aplicación.

En cuanto al núcleo 'C64x+', la posibilidad de emplear la utilidad *Memory Browser* de CCS ha permitido ver someramente los datos copiados de la imagen descodificada por OpenSVC en la segunda *pool* de CMEM, facilitando el trabajo realizado.

5.2.5 Obtención del tamaño de NAL para 'ARM' y 'C64x+'

El principal objetivo de esta prueba fue corroborar el tamaño de cada una de las NALs pertenecientes al paquete de datos codificado y copiado en la primera *pool* de CMEM por parte del núcleo 'ARM'. Para ello, se mantuvo el proceso de descodificación del reproductor MPlayer, alojado en el núcleo 'ARM', donde se identificó el tamaño de cada NAL y posteriormente se realizó la misma operación en el núcleo 'C64x+' con el descodificador OpenSVC.

Esta identificación del tamaño de cada NAL permitió detectar uno de los fallos que más esfuerzo costó solventar: la correcta copia de datos en la primera *pool* de CMEM hasta una determinada cantidad. Como se indicó en el apartado 3.2.2.2, CMEM permite almacenar los datos en alguna de las *pools* declaradas en la configuración del módulo o bien en memoria dinámica (*heap*). En las etapas iniciales de desarrollo se desconocía la ubicación de los datos manejados por el módulo, resultando ser la memoria dinámica el lugar elegido, de tal forma que la identificación de las NAL en el núcleo 'C64x+' se realizaba correctamente pero hasta un determinado número de NALs, momento en el cual los tamaños obtenidos en ambos núcleos diferían completamente.

Tras numerosas pruebas, entre ellas la comentada en el apartado anterior a través de la función `memcpm` o la creación de más *pools* en CMEM para realizar comprobaciones entre los datos copiados, se localizó la fuente de error. Una rápida modificación en el código (ver apartado 4.3.2) permitió ubicar los datos en las *pools* en vez de en memoria dinámica.

Otro problema asociado al manejo de datos entre los dos núcleos del procesador OMAP3530 y que fue detectado mediante la identificación del tamaño de cada NAL fue la acción de la memoria caché. En la aplicación desarrollada en este Trabajo Fin de Máster, tanto 'ARM' como 'C64x+' acceden para leer y escribir datos a la memoria compartida CMEM, provocando que parte de los datos sean almacenados en memoria caché. Esto hace que en posteriores lecturas de paquetes se obtenga una versión que tenga datos del paquete actual con datos del anterior paquete procesado, provocando que el contenido del paquete actual sea erróneo.

Ya que las memorias cachés deben estar operativas para optimizar la descodificación de imágenes en el 'C64x+', se realiza en este núcleo una invalidación de la memoria caché por cada paquete recibido. En concreto, esa invalidación será de tantos datos como *bytes* tenga el anterior paquete procesado (ver apartado 4.3.5, Fig. 77). De este modo se consigue solucionar el problema asociado a la memoria caché sin descartarlas por completo.

Comprobada la correcta identificación del tamaño de cada NAL en el descodificador OpenSVC alojado en el 'C64x+', se elimina el proceso de descodificación en el núcleo 'ARM' mencionado en la introducción de este subapartado con el fin de reducir la carga computacional de éste.

5.2.6 Salvado de imágenes a disco

La posibilidad de guardar imágenes descodificadas a un fichero de disco tanto en el 'ARM' como en el 'C64x+' supuso una nueva alternativa a la comentada en el apartado 5.2.4 para comprobar los contenidos de las *pools* de CMEM, en concreto la segunda de ellas, destinada a almacenar la información de las imágenes descodificacadas.

Dada la mala representación que se hacía de la imagen en el monitor a través del controlador de salida de vídeo *fbdev*, se realizaron una serie de pasos basados en el salvado de imágenes a un fichero para su posterior análisis[30].

1. Salvado a disco de la imagen descodificada por OpenSVC en el núcleo 'C64x+' (ver Fig. 94).
2. Adaptación de la información perteneciente a la segunda *pool* de CMEM en el núcleo 'ARM' (ver apartado 4.4.2).
3. Salvado a disco de la imagen descodificada y copiada en CMEM en el núcleo 'ARM'.
4. Comparación de ambas imágenes mediante visores de imágenes para corroborar la descodificación y posterior transferencia de información entre núcleos.

```
#ifdef SAVEFILE_h264_c
FILE *fichss;
int g;
int luz = avctx->imagen_datos_AVCodecContext;
int primerCr = (luz + (avctx->dimension_x_AVCodecContext+32)*(avctx->dimension_y_AVCodecContext));
int segundoCr = (primerCr + ((avctx->dimension_x_AVCodecContext+32)*(avctx->dimension_y_AVCodecContext)/4));

fichss = fopen("Sincasting_fileh264_line3460.yuv", "wb+"); //Modo escritura

if(!fichss)
{
    printf("\nError al abrir fichero fichss.\n");
}
else
{
    for(g=0; g<(avctx->dimension_y_AVCodecContext); g++)
    {
        fwrite(luz + g * (avctx->dimension_x_AVCodecContext+32), 1, (avctx->dimension_x_AVCodecContext+32), fichss);
    }
    for(g=0; g<(avctx->dimension_y_AVCodecContext/2); g++)
    {
        fwrite(primerCr + g * ((avctx->dimension_x_AVCodecContext+32)/2), 1, ((avctx->dimension_x_AVCodecContext+32)/2), fichss);
    }
    for(g=0; g<(avctx->dimension_y_AVCodecContext/2); g++)
    {
        fwrite(segundoCr + g * ((avctx->dimension_x_AVCodecContext+32)/2), 1, ((avctx->dimension_x_AVCodecContext+32)/2), fichss);
    }

    fclose(fichss);
}//Llave else de apertura de fichero

#endif
```

Fig. 94 Sección de código que guarda la imagen descodificada a disco en el núcleo 'ARM'

Sin embargo, a través de las visores *software* YUV Player [41] y 7YUV [42] no se encontró el motivo por el cual la representación no se hacía correctamente. Fue mediante el *software* de

[30] Se empleo una estructura de compilación condicionada para el salvado de imágenes a disco, pudiendo rápidamente habilitar/deshabilitar esa sección de código a través de la constante SAVEFILE_h264_c.

edición Ultraedit [43], donde se descubrió el problema asociado a los bordes de la imagen descodificada. Tal y como se comentó en el apartado 4.3.4, el estándar de codificación H.264/SVC presenta la posibilidad de realizar predicciones de tamaño macrobloque (16x16) más allá de los límites de la imagen. Esto supone incrementar la cantidad de datos copiados por cada línea en 32 *bytes*, que se corresponden con los 16 del último macrobloque de la fila *n* (borde derecho) y con los restantes 16 del primer macrobloque de la fila *n+1* (borde izquierdo).

Esta característica del estándar no se contempló inicialmente en el salvado de la imagen a disco en el núcleo 'C64x+' pero sí en el 'ARM' de forma que analizando ambos ficheros a través de Ultraedit se observaron diferencias únicamente de 32 *bytes* por línea entre dos imágenes, los correspondientes a los bordes de la imagen. Detectada la fuente de error, se modificó el código perteneciente a la copia de datos de la imagen (núcleo 'C64x+') en la segunda *pool* de CMEM para realizar la presentación de imágenes en el monitor correctamente.

5.2.7 Reinicio del sistema embebido BeagleBoard

Uno de los mayores hándicaps en la realización de este trabajo ha sido el reinicio de la tarjeta BeagleBoard (proceso superior a dos minutos) tras cada depuración de la aplicación desarrollada. Esto se debe a que DSPLink deja a cargo del usuario la liberación de recursos (*clean up*) y la desconexión ordenada de los núcleos del OMAP3530. De momento no se ha implementado ninguna rutina que permita no reiniciar el sistema tras cada prueba, pero sí que se ha localizado una referencia [44] que soluciona este importante problema y que se llevará a cabo en trabajos futuros.

5.3 Caracterización del descodificador

En este apartado se presentará el rendimiento perteneciente al proceso de descodificación de secuencias de vídeo escalables ejecutándose en el núcleo 'C64x+'. Para ello, se proporciona previamente el procedimiento de medida realizado así como algunas características asociadas a la frecuencia de funcionamiento del 'C64x+'.

5.3.1 Procedimiento de medida

Para evaluar el rendimiento del descodificador se han empleado funciones de temporización [45] pertenecientes al sistema operativo DSP/BIOS. En concreto, la función `CLK_gethtime()`, la cual captura el valor de un temporizador de alta resolución que proporciona el sistema operativo; y la función `CLK_countspms()` que proporciona información de la frecuencia de trabajo del 'C64x+'. Asociado al tema de la frecuencia, una vez más es el núcleo 'ARM' el que controla la frecuencia de funcionamiento del 'C64x+', ofreciendo tres posibles frecuencias de trabajo para el núcleo 'C64x+', dependiente cada una de ellas de la frecuencia a la que opere el 'ARM'. La siguiente tabla (ver Tabla 17) muestra la relación entre frecuencias.

Modo de trabajo	Frecuencia de funcionamiento	
	ARM	DSP
1	720 MHz	520 MHz
2	600 MHz	430 MHz
3	550 MHz	400 MHz

Tabla 17 Relación de frecuencias entre los núcleos del procesador OMAP3530

La frecuencia del 'ARM' sí que se ha modificado dentro de los diferentes modos de trabajo mostrados en la tabla, seleccionando el primero de ellos, ya que en teoría garantiza la frecuencia de funcionamiento más elevada para el núcleo 'C64x+'. La Fig. 95 muestra el ajuste realizado [46] en un terminal de consola del sistema operativo Linux embebido en el núcleo 'ARM'.

```
root@beagleboard:~# cd /sys/devices/system/cpu/cpu0/cpufreq/
root@beagleboard:/sys/devices/system/cpu/cpu0/cpufreq# ls
affected_cpus              related_cpus                scaling_governor
cpuinfo_cur_freq           scaling_available_frequencies  scaling_max_freq
cpuinfo_max_freq           scaling_available_governors    scaling_min_freq
cpuinfo_min_freq           scaling_cur_freq            scaling_setspeed
cpuinfo_transition_latency  scaling_driver              stats
root@beagleboard:/sys/devices/system/cpu/cpu0/cpufreq# cat *
0
720000
720000
125000
300000
0
720000 600000 550000 500000 250000 125000
conservative ondemand userspace powersave performance
root@beagleboard:/sys/devices/system/cpu/cpu0/cpufreq# cat cpuinfo_cur_freq
720000
root@beagleboard:/sys/devices/system/cpu/cpu0/cpufreq#
```

Fig. 95 Selección de la máxima frecuencia de funcionamiento en el núcleo 'ARM'

Sin embargo, pese a la modificación de la frecuencia en el 'ARM', no se ha conseguido trabajar a una frecuencia superior a 360 MHz en el núcleo 'C64x+', valor obtenido a través de la función `CLK_countspms()`.

Por tanto, fijando una frecuencia de trabajo de 360 MHz para el 'C64x+', el uso de la función `CLK_gethtime()` en determinadas posiciones del código del descodificador ha permitido calcular el número de ciclos de reloj empleado en las tareas a medir. En concreto, se han realizado tres medidas para cada paquete procesado, eliminando en cada una de ellas el tiempo empleado en copiar la imagen descodificada a la segunda *pool* de CMEM (en caso de que el paquete contuviera una imagen). El diagrama (ver Fig. 96) muestra la ubicación de las medidas dentro del proceso de descodificación en el núcleo 'C64x+'.

De este modo, se toma la primera medida nada más recibir el mensaje procedente del núcleo 'ARM' (denominada en el diagrama *CAP1*). A continuación se lleva a cabo todo el proceso de obtención y procesamiento de unidades NAL de un paquete de datos codificado, donde es necesario diferenciar entre dos posibles situaciones: que el procesamiento de las NALs del paquete genere como resultado la descodificación de una imagen o que se haya procesado otro tipo de unidad NAL, como SPS (*Sequence Parameter Set*), PPS (*Processing Parameter Set*), SEI (*Supplemental Enhancement Information*), etc.

Esta distinción provoca que en el primero de los casos se tome una nueva medida (denominada en el diagrama *CAP2*) a través de la función `CLK_gethtime()` con el fin de eliminar posteriormente el tiempo asociado a la copia de la imagen en la segunda *pool* de CMEM. Si por el contrario no se ha descodificado una imagen, esta medida, *CAP2*, se omite. Finalmente, procesados todos los datos del paquete, se hace uso nuevamente de la función `CLK_gethtime()` para tomar la tercera y última de las medidas (denominada en el diagrama *CAP3*) y se envía el mensaje al núcleo 'ARM' para continuar la ejecución de la aplicación.

Una vez que el 'C64x+' finaliza la comunicación con el 'ARM' por cada paquete procesado, se realizan los cálculos necesarios a través de las tres medidas tomadas (dos en caso de no haber guardado la imagen). Para llevar el control total del tiempo invertido, se cuenta con una variable (denominada en el gráfico *Acumulado*) común a las dos situaciones presentadas (imagen descodificada u otras unidades NAL). Realizadas las operaciones aritméticas en función del resultado de la descodificación, se comprueba si se han descodificado un total de 100 imágenes, parámetro fijado para obtener el número medio de ciclos de reloj por imagen. En caso positivo, se da por terminada la medida para una secuencia de vídeo escalable determinada, mientras que si no se ha alcanzado aún ese valor, se repetirá el procedimiento descrito hasta lograrlo.

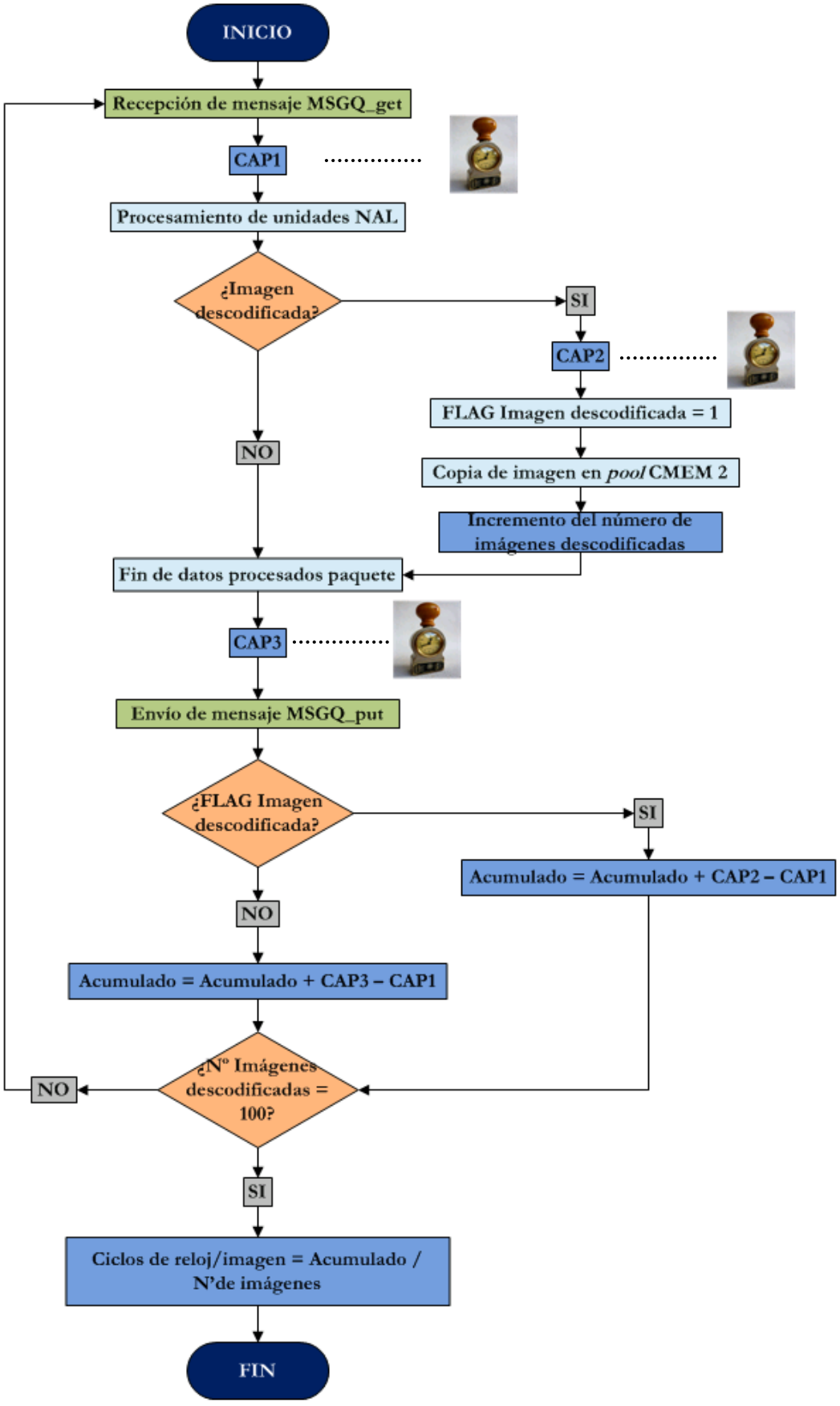

Fig. 96 Procedimiento de medida en el descodificador OpenSVC

5.3.2 Secuencias para evaluar el rendimiento del descodificador OpenSVC

En este apartado se presentarán las diferentes secuencias de vídeo escalable empleadas para evaluar el rendimiento del descodificador OpenSVC alojado en el núcleo 'C64x+' del procesador OMAP3530. En concreto, se han utilizado cinco secuencias de vídeo (*Akiyo*, *Coastguard*, *Foreman*, *Mobile* y *News*) que fueron codificadas mediante un codificador H.264/SVC comercial [47].

Los principales parámetros de codificación empleados para obtener las diferencias secuencias de prueba son los siguientes:

- Tamaño del GOP (*Group of Picture*) es igual a ocho cuadros progresivos.
- Codificación de entropía CABAC (*Context-adaptive binary arithmetic coding*).
- Activación del filtro antibloques (*deblocking*).
- Tres imágenes de referencia para predicciones *inter*.
- Imagen codificada de tipo B por cada imagen tipo I.
- Permitidas todas las particiones de macrobloques disponibles en la norma [25].

Además, cada una de las cinco secuencias presentadas contendrá a su vez seis capas (escalabilidades) de las ocho combinaciones posibles:

- Dos resoluciones espaciales (QCIF y CIF[31]),
- Dos tasas de imágenes distintas (12,5 y 25 imágenes por segundo) y
- Dos calidades (alta y baja).

En cuanto al *bitrate* de cada secuencia, éste será de 512 Kbps, estando codificada la capa base con un *bitrate* de 102 Kbps, esto es, un 20 % de la tasa total. Con el fin de evaluar la influencia que tienen las capas de una misma secuencia en el rendimiento del descodificador, se han generado dos tipos diferentes de secuencias de prueba.

La estructura del primer conjunto de secuencias de pruebas se muestra en la Fig. 97, donde las dos escalabilidades temporales mencionadas se omiten. En este conjunto de secuencias la primera capa de mejora se deriva de la capa base aumentando la calidad de la misma (escalabilidad de calidad), mientras que la segunda capa también se deriva de las anteriores aportando en este caso una mejora que aumenta la resolución espacial (escalabilidad espacial). Este tipo de secuencias se denominará "Calidad + Espacial".

Fig. 97 Secuencias "Calidad + Espacial" con la escalabilidad temporal omitida

[31] CIF define un secuencia de vídeo con una resolución de 352x288 pixels, mientras que QCIF (Quarter CIF) define una resolución de 176x144 pixels.

Por otro lado, la Fig. 98 ofrece el segundo conjunto de secuencias de prueba, donde nuevamente las capas temporales no se han representado. Ahora, la primera capa de mejora se corresponde con un aumento de la resolución espacial (escalabilidad espacial) mientras que la siguiente capa de mejora incorpora una mejora de calidad sobre las capas anteriores (escalabilidad de calidad). Este tipo de secuencias se denominará "Espacial + Calidad".

Fig. 98 Secuencias "Espacial + Calidad" con la escalabilidad temporal omitida

5.3.3 Resultados obtenidos

Para la realización de este apartado se ha medido el rendimiento del descodificador en tres escenarios diferentes, todos ellos empleando el sistema embebido BeagleBoard basado en el procesador OMAP3530.

Se definen a continuación:

1. Descodificador OpenSVC aislado y optimizado ejecutándose en el núcleo 'C64x+' [3][48]. Se denominará C64x+ Aislado.
2. Descodificador OpenSVC integrado en el reproductor MPlayer y ejecutándose en el núcleo 'C64x+'. Se denominará C64x+ MPlayer.
3. Descodificador OpenSVC integrado en el reproductor MPlayer y ejecutándose en el núcleo 'ARM' [1]. Se denominará ARM MPlayer.

Excepto el primero de los escenarios descritos donde se han tomado los datos obtenidos en [3] como referencia, las medidas pertenecientes a los escenarios numerados como 2 y 3 han sido realizadas íntegramente en este Trabajo Fin de Máster.

En todos los escenarios, el rendimiento del descodificador ha sido medido después de descodificar 100 imágenes, tal y como se representó en la Fig. 96. Sin embargo, el procedimiento de medida difiere notablemente. En el caso del núcleo 'C64x+', se emplearon funciones pertenecientes al sistema operativo DSP/BIOS, tal y como se indicó en el apartado 5.3.1, mientras que en el caso del escenario 3, el rendimiento en el núcleo 'ARM' fue medido a través de la utlidad *top* de Linux (ver Fig. 99), esto es, un comando ejecutado en un terminal de consola del sistema operativo embebido en el 'ARM' que muestra a tiempo real un listado de los procesos que se están ejecutando en el sistema, especificando además el porcentaje de CPU y memoria empleados, sus IDs, usuarios que lo están ejecutando, etc.

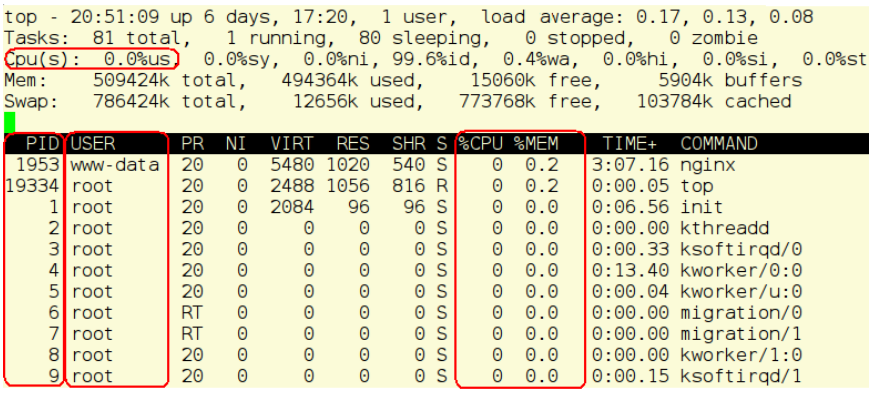

Fig. 99 Comando *top* en sistemas operativos Linux

Presentados los diferentes escenarios bajo los cuales se analizará el rendimiento del descodificador OpenSVC, se describirán a continuación los dos tipos de medidas realizadas.

- Porcentaje de CPU necesario para lograr la descodificación del vídeo escalable en tiempo real (ver apartado 5.3.3.1).
- Número de imágenes por segundo (*frames per second, fps*) si el porcentaje de CPU fuera del 100 %, es decir, si el núcleo ('ARM' o 'C64x+') estuviera dedicado a esta tarea exclusivamente (ver apartado 5.3.3.2).

Para ambas medidas se han analizado las diferentes capas (escalabilidades) de los dos tipos de secuencias presentadas en el apartado 5.3.2, denominadas como "Espacial + Calidad" y "Calidad + Espacial".

En relación a los resultados obtenidos, es muy importante destacar la frecuencia de trabajo a la cual opera cada núcleo, ya que la relación de frecuencias entre 'ARM' y 'C64x+' es 2:1, esto es, el 'ARM' trabaja a 720 MHz, duplicando la frecuencia del núcleo 'C64x+', la cual es de 360 MHz. Como se analizará más adelante, una vez mostrados los resultados, existen casos en los que aparentemente el 'ARM' obtiene mejores resultados que la integración entre ambos núcleos realizada en este Trabajo Fin de Máster, pero hay que tener en cuenta la relación de frecuencias recién comentada.

En cuanto a la terminología empleada para cada capa de una secuencia, se seguirá el siguiente formato:

- Nombre de la secuencia.
- Resolución espacial: QCIF o CIF.
- Calidad, Q: *High* o *Low*.
- Número de imágenes, T: 12,5 o 25.

A continuación se presentarán los resultados obtenidos para los dos tipos de medidas a realizar: porcentaje de CPU para lograr la descodificación en tiempo real (Tabla 18 y Tabla 19) y número de imágenes por segundo si el porcentaje de CPU fuera del 100 % (Tabla 21 y Tabla 22). En ambos casos el orden de presentación de las secuencias será "Calidad + Espacial" y "Espacial + Calidad".

5.3.3.1 Porcentaje de CPU para procesamiento en tiempo real

La Tabla 18 muestra el porcentaje de CPU necesario para descodificar cada capa de las secuencias "Calidad + Espacial" en cada uno de los diferentes escenarios descritos anteriormente (C64x+ Aislado, C64x+ MPlayer, ARM MPlayer) indicando la frecuencia de funcionamiento de cada núcleo del procesador OMAP3530. Las medidas fueron calculadas después de descodificar 100 imágenes.

Secuencia/Procesador	Porcentaje de CPU (%)		
	C64x+ Aislado	C64x+ MPlayer	ARM MPlayer
Secuencias "Calidad + Espacial"	360 MHz	360 MHz	720 MHz
Akiyo QCIF, Q=Low, T=12,5	6,51	9,70	8,54
Akiyo QCIF, Q=Low, T=25	13,53	19,89	17,08
Akiyo QCIF, Q=High, T=12,5	13,77	22,92	25,00
Akiyo QCIF, Q=High, T=25	30,98	50,14	50,00
Akiyo CIF, Q=High, T=12,5	36,14	62,76	86,67
Akiyo CIF, Q=High, T=25	84,16	138,34	173,33
Coastguard QCIF, Q=Low, T=12,5	6,64	9,71	8,85
Coastguard QCIF, Q=Low, T=25	14,03	20,68	17,71
Coastguard QCIF, Q=High, T=12,5	13,85	21,56	24,69
Coastguard QCIF, Q=High, T=25	31,74	51,36	49,38
Coastguard CIF, Q=High, T=12,5	37,29	59,57	85,94
Coastguard CIF, Q=High, T=25	86,57	142,51	171,88
Foreman QCIF, Q=Low, T=12,5	6,78	10,26	10,21
Foreman QCIF, Q=Low, T=25	14,25	21,19	20,42
Foreman QCIF, Q=High, T=12,5	13,86	23,30	25,52
Foreman QCIF, Q=High, T=25	31,84	51,79	51,04
Foreman CIF, Q=High, T=12,5	36,71	65,61	87,08
Foreman CIF, Q=High, T=25	86,11	143,63	174,17
Mobile QCIF, Q=Low, T=12,5	6,92	10,36	10,21
Mobile QCIF, Q=Low, T=25	14,07	20,82	19,17
Mobile QCIF, Q=High, T=12,5	14,17	23,40	25,52
Mobile QCIF, Q=High, T=25	31,98	51,82	49,79
Mobile CIF, Q=High, T=12,5	37,62	66,53	87,08
Mobile CIF, Q=High, T=25	86,00	144,02	174,38
News QCIF, Q=Low, T=12,5	6,40	9,96	9,58
News QCIF, Q=Low, T=25	13,61	20,47	17,71
News QCIF, Q=High, T=12,5	13,55	22,78	24,90
News QCIF, Q=High, T=25	30,44	49,02	49,58
News CIF, Q=High, T=12,5	36,29	59,35	87,19
News CIF, Q=High, T=25	82,59	136,63	173,54

Tabla 18 Porcentaje de CPU para procesamiento en tiempo real - Secuencias "Calidad + Espacial"

La Tabla 19 muestra el porcentaje de CPU necesario para descodificar cada capa de las secuencias "Espacial + Calidad" en cada uno de los diferentes escenarios descritos anteriormente (C64x+ Aislado, C64x+ MPlayer, ARM MPlayer) indicando la frecuencia de funcionamiento de cada núcleo del procesador OMAP3530. Las medidas fueron calculadas después de descodificar 100 imágenes.

Secuencia/Procesador	Porcentaje de CPU (%)		
	C64x+ Aislado	C64x+ MPlayer	ARM MPlayer
Secuencias "Espacial + Calidad	360 MHz	360 MHz	720 MHz
Akiyo QCIF, Q=Low, T=12,5	6,13	9,33	8,25
Akiyo QCIF, Q=Low, T=25	13,23	19,10	17,13
Akiyo CIF, Q=Low, T=12,5	28,41	49,31	57,08
Akiyo CIF, Q=Low, T=25	68,34	109,81	114,17
Akiyo CIF, Q=High, T=12,5	51,55	89,54	86,77
Akiyo CIF, Q=High, T=25	122,65	203,87	173,54
Coastguard QCIF, Q=Low, T=12,5	6,62	9,76	9,23
Coastguard QCIF, Q=Low, T=25	13,97	20,55	17,58
Coastguard CIF, Q=Low, T=12,5	30,97	51,53	58,13
Coastguard CIF, Q=Low, T=25	70,88	115,86	116,25
Coastguard CIF, Q=High, T=12,5	52,95	93,74	86,67
Coastguard CIF, Q=High, T=25	1269	211,76	173,33
Foreman QCIF, Q=Low, T=12,5	6,64	10,00	10,25
Foreman QCIF, Q=Low, T=25	14,27	21,10	20,34
Foreman CIF, Q=Low, T=12,5	30,54	52,04	62,60
Foreman CIF, Q=Low, T=25	70,60	117,98	125,21
Foreman CIF, Q=High, T=12,5	52,71	93,16	87,08
Foreman CIF, Q=High, T=25	126,40	211,45	174,38
Mobile QCIF, Q=Low, T=12,5	6,62	10,07	9,79
Mobile QCIF, Q=Low, T=25	13,96	20,48	19,20
Mobile CIF, Q=Low, T=12,5	31,58	52,71	62,60
Mobile CIF, Q=Low, T=25	71,00	116,96	126,04
Mobile CIF, Q=High, T=12,5	54,01	96,15	87,19
Mobile CIF, Q=High, T=25	125,51	210,49	173,54
News QCIF, Q=Low, T=12,5	6,23	9,18	9,05
News QCIF, Q=Low, T=25	13,31	19,53	18,76
News CIF, Q=Low, T=12,5	29,16	51,79	63,02
News CIF, Q=Low, T=25	68,77	110,01	116,25
News CIF, Q=High, T=12,5	50,95	89,13	86,77
News CIF, Q=High, T=25	124,10	202,84	174,17

Tabla 19 Porcentaje de CPU para procesamiento en tiempo real - Secuencias "Espacial + Calidad"

Respecto a los datos mostrados en la Tabla 18, en primer lugar resaltar que únicamente se alcanza el tiempo real en todas las capas para la versión aislada y optimizada del descodificador OpenSVC ejecutándose en el núcleo 'C64x+'. Para los otros dos escenarios, C64x+ MPlayer y ARM MPlayer, la última capa de mejora (resolución espacial CIF, nivel

de calidad alto, *high* y una tasa de 25 imágenes por segundo) es la que impide el funcionamiento el tiempo real de todas las capas correspondientes a las diferentes secuencias de prueba empleadas.

Por el contrario, para los datos mostrados en la Tabla 19 no se consigue el funcionamiento en tiempo real en ninguno de los tres escenarios. A la vista de los resultados, y contrastando la información de ambas tablas, se observa claramente que el rendimiento es mayor si la primera capa de mejora (obviando las temporales) es un incremento en la calidad de la secuencia en lugar de una capa de mejora espacial. Las Fig. 100 y Fig. 101 muestran una de las posibles comparativas, tomando como ejemplo la secuencia *Coastguard* en la cual se refleja perfectamente el comportamiento descrito.

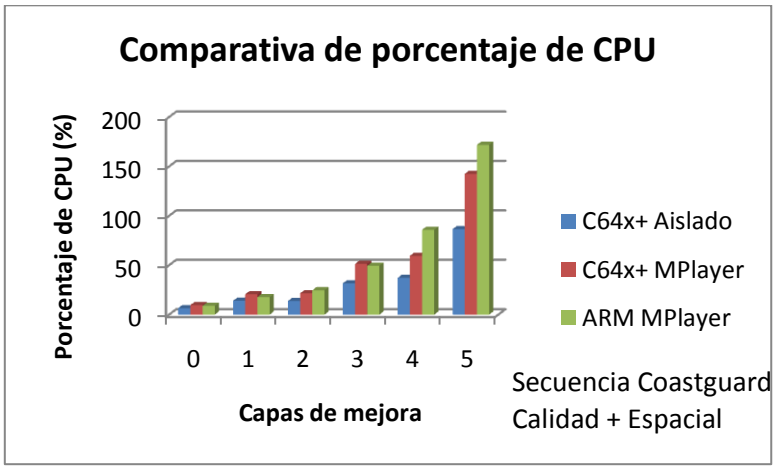

Fig. 100 Comparativa de porcentaje de CPU para la secuencia *Coastguard* (Calidad + Espacial)

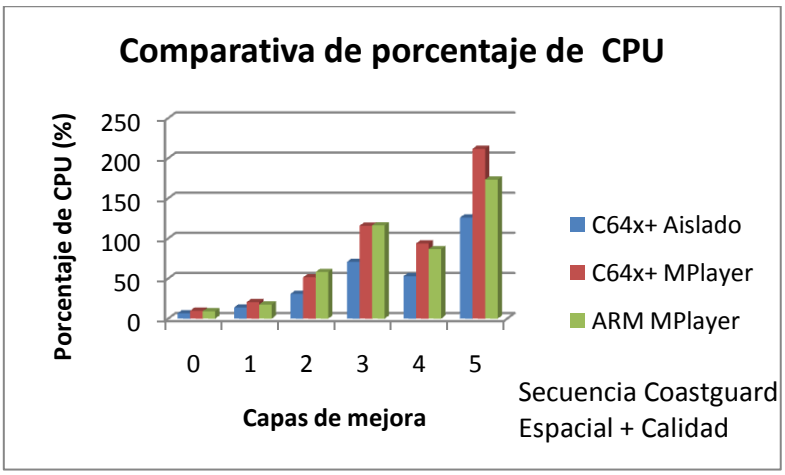

Fig. 101 Comparativa de porcentaje de CPU para la secuencia *Coastguard* (Espacial + Calidad)

Si ahora se analizan los resultados tomando como referencia el escenario donde se lleva a cabo la descodificación de la secuencia de vídeo escalable, se demuestra que la versión optimizada y aislada del descodificador OpenSVC ejecutada en el núcleo 'C64x+' presenta unos resultados muchos mejores en cuanto a porcentaje de CPU empleado respecto a los otros escenarios. En el caso de la integración llevada a cabo en este Trabajo Fin de Máster, C64x+ MPlayer, existe un incremento de carga computacional para todas las secuencias ya

que se trata de una primera versión en la cual el grado de optimización es mínimo. Ambos escenarios basados en el núcleo 'C64x+' trabajan a la misma frecuencia, 360 MHz.

Por el contrario, el tercer escenario, ARM MPlayer, presenta una frecuencia de funcionamiento de 720 MHz, duplicando la del núcleo 'C64x+', lo que podría hacer pensar en una primera aproximación en la obtención de unos resultados mejores que los dos primeros escenarios. Sin embargo, esos resultados son, en la mayoría de las capas, inferiores a los alcanzados en el segundo de los escenarios (C64x+ MPlayer) tanto para secuencias "Calidad + Espacial" como "Espacial + Calidad", poniendo de manifiesto la escasa optimización de este tipo de arquitecturas para tareas de descodificación de vídeo.

Dada la similitud de los resultados obtenidos en una capa determinada para todas las secuencias, se presenta a continuación una comparativa (ver Tabla 20) que relaciona el valor medio para cada capa calculado del siguiente modo:

$$\left(\text{Valor medio}_{\text{capa x}}\right)(\%) = \frac{\left(\%\text{CPU Akiyo}_{\text{capa x}}\right) + \left(\%\text{CPU Mobile}_{\text{capa x}}\right) + \ldots + \left(\%\text{CPU News}_{\text{capa x}}\right)}{\text{Número de secuencias}}$$

La comparativa se realizará entre el descodificador OpenSVC ejecutándose en el núcleo 'C64x+' e integrado en MPlayer y la versión que se ejecuta en el 'ARM'.

Secuencia/Procesador	Valor medio de CPU (%)	
	C64x+ MPlayer	ARM MPlayer
Número de capa para secuencias "Calidad + Espacial"	360 MHz	720 MHz
Capa 0, QCIF, Q=Low, T=12,5	10,00	9,48
Capa 1, QCIF, Q=Low, T=25	20,61	18,42
Capa 2, QCIF, Q=High, T=12,5	22,79	25,12
Capa 3, QCIF, Q=High, T=25	50,83	49,96
Capa 4, CIF, Q=High, T=12,5	62,76	86,79
Capa 5, CIF, Q=High, T=25	141,03	173,46
Número de capa para secuencias "Espacial + Calidad"		
Capa 0, QCIF, Q=Low, T=12,5	9,67	9,31
Capa 1, QCIF, Q=Low, T=25	20,15	18,60
Capa 2, CIF, Q=Low, T=12,5	51,48	60,69
Capa 3, CIF, Q=Low, T=25	114,13	119,58
Capa 4, CIF, Q=High, T=12,5	92,34	86,90
Capa 5, CIF, Q=High, T=25	208,08	173,79

Tabla 20 Comparativa del valor medio por capa entre los escenarios C64x+MPlayer y ARM

Es importante destacar esta mejora significativa del escenario dos (C64x+ MPlayer) respecto al escenario 3 (ARM MPlayer) ya que trabajar a la mitad de frecuencia implica un notable decremento en el consumo de energía total del procesador OMAP3530, factor crítico para dispositivos portátiles. Por tanto, pese a que los resultados no sean los esperados respecto a la versión aislada y optimizada del descodificador, el desarrollo realizado en este Trabajo Fin de Máster ha reducido significativamente la carga computacional del 'ARM', lo que implica a su vez una reducción del consumo de energía del terminal y una mayor disposición del propio 'ARM' para la ejecución de otras tareas.

5.3.3.2 Número de imágenes por segundo con CPU dedicada al 100%

La Tabla 21 muestra el número de imágenes por segundo descodificadas si el porcentaje de CPU fuera del 100 % para capa de las secuencias "Calidad + Espacial" en cada uno de los diferentes escenarios descritos anteriormente (C64x+ Aislado, C64x+ MPlayer, ARM MPlayer) indicando la frecuencia de funcionamiento de cada núcleo del procesador OMAP3530.

Secuencia/Procesador	FPS al 100%		
	C64x+ Aislado	C64x+ MPlayer	ARM MPlayer
Secuencias "Calidad + Espacial"	360 MHz	360 MHz	720 MHz
Akiyo QCIF, Q=Low, T=12,5	384,62	257,66	292,68
Akiyo QCIF, Q=Low, T=25	185,19	125,72	146,37
Akiyo QCIF, Q=High, T=12,5	182,48	109,05	100,00
Akiyo QCIF, Q=High, T=25	80,91	49,86	50,00
Akiyo CIF, Q=High, T=12,5	69,25	39,83	28,85
Akiyo CIF, Q=High, T=25	29,73	18,07	14,42
Coastguard QCIF, Q=Low, T=12,5	378,79	257,41	282,35
Coastguard QCIF, Q=Low, T=25	178,57	120,90	141,16
Coastguard QCIF, Q=High, T=12,5	181,16	115,93	101,27
Coastguard QCIF, Q=High, T=25	78,86	48,67	50,63
Coastguard CIF, Q=High, T=12,5	67,20	41,97	29,09
Coastguard CIF, Q=High, T=25	28,90	17,54	14,55
Foreman QCIF, Q=Low, T=12,5	373,13	243,70	244,90
Foreman QCIF, Q=Low, T=25	176,06	117,98	122,43
Foreman QCIF, Q=High, T=12,5	181,16	107,31	97,96
Foreman QCIF, Q=High, T=25	78,62	48,27	48,98
Foreman CIF, Q=High, T=12,5	68,12	38,11	28,71
Foreman CIF, Q=High, T=25	29,04	17,41	14,35
Mobile QCIF, Q=Low, T=12,5	362,32	241,23	244,90
Mobile QCIF, Q=Low, T=25	178,57	120,08	130,41
Mobile QCIF, Q=High, T=12,5	177,30	106,82	97,96
Mobile QCIF, Q=High, T=25	78,37	48,25	50,21
Mobile CIF, Q=High, T=12,5	66,49	37,58	28,71
Mobile CIF, Q=High, T=25	29,07	17,36	14,34
News QCIF, Q=Low, T=12,5	390,63	250,98	260,87
News QCIF, Q=Low, T=25	183,82	122,12	141,16
News QCIF, Q=High, T=12,5	185,19	109,73	100,42
News QCIF, Q=High, T=25	82,24	51,00	50,42
News CIF, Q=High, T=12,5	69,06	42,12	28,67
News CIF, Q=High, T=25	30,30	18,30	14,41

Tabla 21 Número imágenes por segundo descodificadas con CPU al 100 % - Secuencias "Calidad + Espacial"

La Tabla 22 muestra el número de imágenes por segundo descodificadas si el porcentaje de CPU fuera del 100 % para capa de las secuencias "Espacial + Calidad" en cada uno de los diferentes escenarios descritos anteriormente (C64x+ Aislado, C64x+ MPlayer, ARM MPlayer) indicando la frecuencia de funcionamiento de cada núcleo del procesador OMAP3530.

Secuencia/Procesador	FPS al 100%		
	C64x+ Aislado	C64x+ MPlayer	ARM MPlayer
Secuencias "Espacial + Calidad"	360 MHz	360 MHz	720 MHz
Akiyo QCIF, Q=Low, T=12,5	409,84	268,08	303,03
Akiyo QCIF, Q=Low, T=25	189,39	130,90	145,94
Akiyo CIF, Q=Low, T=12,5	88,03	50,70	43,80
Akiyo CIF, Q=Low, T=25	36,60	22,77	21,90
Akiyo CIF, Q=High, T=12,5	48,54	27,92	28,81
Akiyo CIF, Q=High, T=25	20,39	12,26	14,41
Coastguard QCIF, Q=Low, T=12,5	378,79	256,24	270,86
Coastguard QCIF, Q=Low, T=25	179,86	121,65	142,21
Coastguard CIF, Q=Low, T=12,5	80,91	48,51	43,01
Coastguard CIF, Q=Low, T=25	35,31	21,58	21,51
Coastguard CIF, Q=High, T=12,5	47,26	26,67	28,85
Coastguard CIF, Q=High, T=25	19,84	11,81	14,42
Foreman QCIF, Q=Low, T=12,5	378,79	249,99	243,90
Foreman QCIF, Q=Low, T=25	176,06	118,50	122,91
Foreman CIF, Q=Low, T=12,5	81,97	48,04	39,93
Foreman CIF, Q=Low, T=25	35,41	21,19	19,97
Foreman CIF, Q=High, T=12,5	47,44	26,83	28,71
Foreman CIF, Q=High, T=25	19,78	11,82	14,34
Mobile QCIF, Q=Low, T=12,5	378,79	248,29	255,36
Mobile QCIF, Q=Low, T=25	179,86	122,09	130,21
Mobile CIF, Q=Low, T=12,5	80,65	47,43	39,93
Mobile CIF, Q=Low, T=25	35,21	21,37	19,83
Mobile CIF, Q=High, T=12,5	46,30	26,00	28,67
Mobile CIF, Q=High, T=25	19,92	11,88	14,41
News QCIF, Q=Low, T=12,5	403,23	272,21	276,24
News QCIF, Q=Low, T=25	187,97	128,04	133,26
News CIF, Q=Low, T=12,5	85,91	48,27	39,67
News CIF, Q=Low, T=25	36,39	22,72	21,51
News CIF, Q=High, T=12,5	49,12	28,05	28,81
News CIF, Q=High, T=25	20,15	12,32	14,35

Tabla 22 Número imágenes por segundo descodificadas con CPU al 100 % - Secuencias "Espacial + Calidad"

Tanto la Tabla 21 como la Tabla 22 corroboran las conclusiones realizadas en el apartado anterior, ya que los datos relativos al número de imágenes descodificadas si la CPU está dedicada exclusivamente a esta tarea se extraen de los cálculos asociados al porcentaje de CPU para procesamiento en tiempo real.

Ese cálculo se realiza del siguiente modo:

$$\text{Millones de ciclos @25fps} = \frac{\text{Porcentaje de CPU(\%)}}{100} \text{ x } \text{Frecuencia}_{\text{CLK}}$$

$$\text{fps @CPU 100 \%} = \frac{\text{Frecuencia}_{\text{CLK}}}{\frac{\text{Millones de ciclos@25fps}}{25}}$$

Por esta razón, el descodificador OpenSVC aislado y optimizado vuelve a presentar los mejores resultados respecto a los otros dos escenarios para los dos tipos de secuencias analizadas. Las Fig. 102 y Fig. 103 muestran para la misma secuencia ejemplificada en el apartado anterior, *Coastguard*, una comparativa del número de imágenes por segundo descodificadas al 100 % de CPU para los dos tipos de secuencias ("Calidad + Espacial" y "Espacial + Calidad") con el fin de verificar los diferentes resultados obtenidos por cada escenario en el apartado anterior.

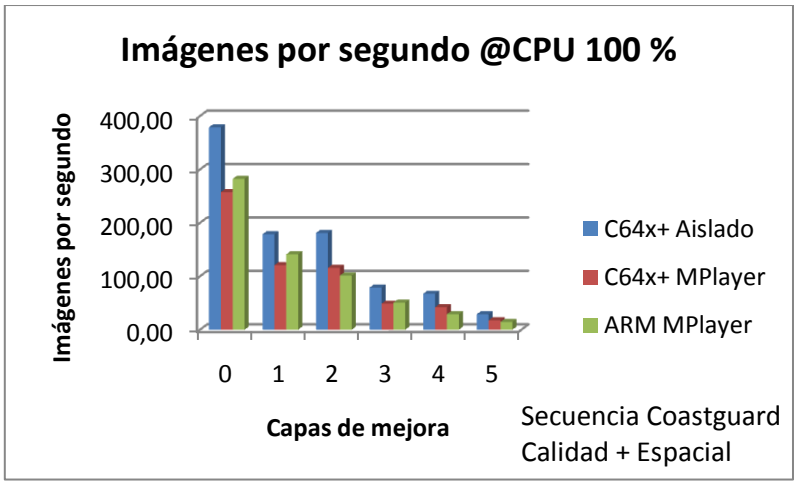

Fig. 102 Imágenes por segundo @ 100 % CPU para la secuencia *Coastguard* (Calidad + Espacial)

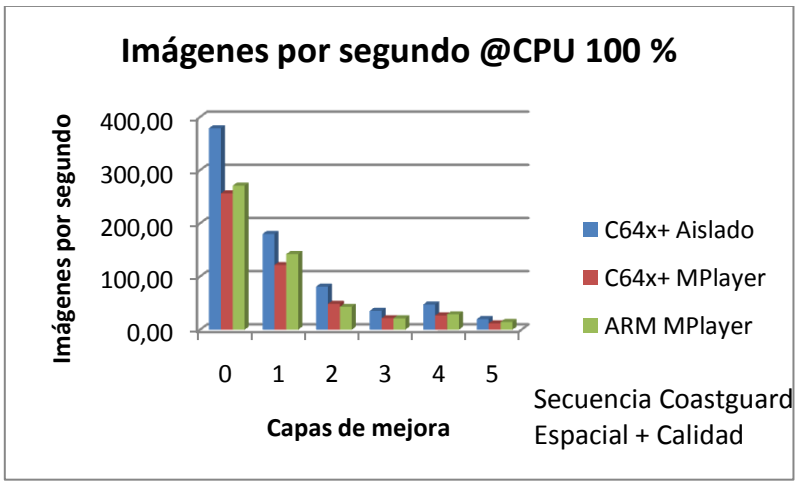

Fig. 103 Imágenes por segundo @ 100 % CPU para la secuencia *Coastguard* (Espacial + Calidad)

Nuevamente se quiere destacar la relación de frecuencias entre los núcleos de procesador OMAP3530, ya que aunque el 'ARM' funcione al doble de velocidad que el 'C64x+', los resultados obtenidos en cuanto a imágenes por segundo al máximo rendimiento de la CPU son en su totalidad similares e incluso peores en algunos casos que los del escenario dos (C64x+MPlayer). Sin embargo, el consumo asociado al procesador OMAP3530 disminuirá notablemente debido a esa relación de frecuencias, siendo este tema, la optimización del consumo en sistemas embebidos, uno de los objetivos prioritarios del trabajo realizado.

6 CONCLUSIONES Y TRABAJOS FUTUROS

En este capítulo se presentan las conclusiones resultantes de la realización de este Trabajo Fin de Máster y las líneas de trabajo futuro.

6.1 Conclusiones

El objetivo principal de este Trabajo Fin de Máster consistía en la integración del reproductor multimedia MPlayer que se ejecuta en el núcleo 'ARM' con el descodificador OpenSVC alojado en el núcleo 'C64x+' con el propósito de distribuir la carga computacional asociada a la descodificación de un vídeo escalable entre los núcleos del procesador OMAP3530, objetivo que se ha logrado en su totalidad.

Para ello, inicialmente se realizó un análisis en profundidad del reproductor MPlayer (con soporte para vídeo escalable) con el fin de conocer su método de trabajo, ya que en todo momento se ha buscado mantener la estructura del reproductor y adaptar el resto de elementos que conforman la aplicación a él. Dada la elevada dependencia y complejidad del reproductor a nivel de estructuras y funciones, ha supuesto un gran esfuerzo comprender el funcionamiento del mismo.

Una vez realizado el análisis del reproductor, se estudió la versión aislada y optimizada del descodificador OpenSVC llevada a cabo en [3], a la cual se le realizaron una serie de cambios significativos en su método de trabajo para adaptarlo a la integración propuesta en este Trabajo Fin de Máster.

A continuación, los módulos DSPLink y CMEM cubrieron aspectos de especial importancia para la sincronización, comunicación y transferencia de información entre los núcleos del procesador OMAP3530. En concreto, el primero de ellos permitió la transferencia de información reducida a través de colas de mensajes así como un método de sincronización mediante esperas bloqueantes entre 'ARM' y 'C64x+'. Además, al presentar el procesador OMAP3530 una jerarquía entre sus núcleos *master-slave*, el 'ARM' tiene control total sobre las acciones a realizar en el 'C64x+', y es precisamente a través de la API de DSPLink donde se ejecutan las funciones de enlace, carga e inicio de la ejecución del programa en el 'C64x+'.

Las limitaciones del núcleo 'ARM' para acceder a una zona de memoria localizada fuera de su espacio de direcciones así como la incapacidad del 'C64x+' para manejar direcciones virtuales y bloques de memoria no contiguos hizo necesario el empleo de una herramienta adicional. CMEM tiene la capacidad de reservar un espacio de memoria contiguo, fuera del espacio de memoria del sistema operativo y hacerlo accesible para a ambos núcleos del procesador OMAP3530. Además, el espacio de memoria reservado por el módulo ofrece la posibilidad de ser dividido en secciones (*pools*) de tamaño variable, de modo que se definieron dos *pools* para separar la copia de datos codificados de las imágenes descodificadas por OpenSVC.

La configuración y obtención de ambos módulos, DSPLink y CMEM, los cuales presentan una relación directa con el mapa de memoria del sistema embebido BeagleBoard, se convirtió en una tarea que generó muchos problemas en las etapas iniciales del desarrollo debido a la compleja configuración del módulo DSPLink. Es más, el mínimo cambio realizado en alguna de las secciones del mapa de memoria obliga a la repetición del proceso de generación del módulo y actualización posterior de librerías y componentes en sus respectivas localizaciones.

Respecto al entorno de desarrollo integrado donde se ha realizado la edición y depuración de aplicaciones para 'ARM' y 'C64x+', se ha empleado Code Composer Studio en su versión 5.1 para Linux, la cual incorpora una nueva técnica de depuración mutiprocesador que ha conseguido sustituir al proceso de depuración laborioso que se venía realizando en el GDEM. Anterior a la realización de este Trabajo Fin de Máster, se requerían dos sistemas operativos diferentes para la depuración de aplicaciones que involucrasen a ambos núcleos: Windows para el 'C64x+' y Linux para 'ARM'. El motivo de esta dualidad se debía a la falta de controladores para los emuladores Blackhawk disponibles en el GDEM en sistemas operativos Linux. Este escenario planteado hacía que el tiempo transcurrido entre la realización de un cambio en el código de cualquier núcleo y la comprobación de que el cambio tenía el efecto deseado fuera superior a tres minutos.

Ahora, esa dualidad de sistemas operativos desaparece, ya que la nueva versión de CCS para Linux si que proporciona soporte para los citados emuladores, por lo que se consigue trabajar bajo una única interfaz para ambos núcleos del procesador OMAP3530. Sin embargo, dada la novedad de la versión empleada, no ha sido fácil descubrir y configurar esa nueva técnica de depuración multiprocesador, ya que la documentación del entorno de Texas Instruments en el comienzo de este Trabajo Fin de Máster era escasa y carente de información asociada a la depuración.

Estudiados los dos elementos principales de este trabajo, MPlayer y OpenSVC, y configurados tanto los módulos DSPLink y CMEM como el entorno de desarrollo CCS, hubo que localizar las secciones de código claves donde ubicar las diferentes funciones que permitirían la integración del reproductor y el descodificador. Tras la realización de esa integración, se ha logrado el siguiente funcionamiento[32]:

1. Identificación de los diferentes *streams* del fichero a descodificar ('ARM').
2. Adaptación de la información por parte del demultiplexor ('ARM').
3. Copia en CMEM (1ª *pool*) de un fragmento de información perteneciente a los datos codificados del fichero.
4. Envío de mensaje a través de la cola de mensajes FIFO (*First In First Out*) implementada por DSPLink ('ARM').
5. Recepción del mensaje y procesamiento de la información codificada por OpenSVC ('C64x+').
6. Copia en CMEM (2ª *pool*) de la imagen descodificada por OpenSVC en caso de haberla ('C64x+').
7. Envío de mensaje a través de la cola de mensajes FIFO implementada por DSPLink ('C64x+').
8. Recepción del mensaje y presentación en un monitor tras previa adaptación de la información de la imagen y del controlador de salida de vídeo ('ARM').
9. Vuelta al punto 3.

Ese funcionamiento se ha detallado completamente en esta memoria, añadiendo todas las pautas y pasos necesarios para lograrlo a través de explicaciones teóricas, secciones de

[32] Entre paréntesis se indica el núcleo del procesador OMAP3530 donde se realiza la tarea.

código comentadas pertenecientes a la aplicación y diagramas de flujo que sintetizan las explicaciones realizadas.

El siguiente paso, una vez realizada con éxito la integración, fue la caracterización del descodificador OpenSVC alojado en el núcleo 'C64x+', definiendo un procedimiento de medida a través de las funciones de temporización del sistema operativo DSP/BIOS. El rendimiento del descodificador ha sido medido para las siguientes cinco secuencias de vídeo escalable: *Akiyo*, *Coastguard*, *Foreman*, *Mobile* y *News*.

Cada secuencia contiene seis capas: dos resoluciones espaciales (QCIF y CIF), dos tasas de imágenes distintas (12,5 y 25 *fps*) y dos calidades (alta y baja). Con el fin de evaluar la influencia que tienen las capas de una misma secuencia en el rendimiento del descodificador, se generaron dos tipos diferentes de secuencias de prueba:

- "Calidad + Espacial", donde la primera capa de mejora se deriva de la capa base aumentado la calidad de la misma, mientras que la segunda capa aporta una mejora en cuanto a la resolución espacial, y,
- "Espacial + Calidad", donde la primera capa de mejora se deriva de la capa base aumentado la resolución espacial de la misma, mientras que la siguiente capa aporta una mejora de calidad.

A continuación se definieron los dos tipos de medidas a realizar para analizar el rendimiento de OpenSVC: porcentaje de CPU para procesamiento en tiempo real y número de imágenes por segundo con dedicación máxima de la CPU. Dados los resultados obtenidos en [3] para la versión aislada y optimizada del descodificador se realizó una comparativa para las medidas descritas entre los siguientes escenarios:

1. Descodificador OpenSVC aislado y optimizado ejecutándose en el núcleo 'C64x+' (F_clk = 360 MHz) [3].
2. Descodificador OpenSVC integrado en el reproductor MPlayer y ejecutándose en el núcleo 'C64x+' (F_clk = 360 MHz).
3. Descodificador OpenSVC integrado en el reproductor MPlayer y ejecutándose en el núcleo 'ARM' (F_clk = 720 MHz) [1].

Los resultados obtenidos determinaron que el escenario 1 logra un mejor rendimiento para todas las medidas realizadas respecto a los otros dos escenarios. La integración realizada en este Trabajo Fin de Máster, numerada como el escenario 2, presenta un incremento de uso de CPU respecto al escenario 1 debido a la falta de optimización, pero aún así garantiza el procesamiento en tiempo real para la mayoría de las capas analizadas.

El aspecto más significativo se presenta en la comparación entre los escenarios 2 y 3, ya que la descodificación realizada en el 'ARM', con una frecuencia de trabajo de 720 MHz, no consigue en la mayoría de las secuencias analizadas igualar los resultados del escenario 2, donde el 'C64x+' trabaja a la mitad de frecuencia, 360 MHz. Esto demuestra que la integración realizada en este Trabajo Fin de Máster ha permitido descargar al 'ARM' de la tarea de la descodificación, liberando una elevada cantidad de carga computacional y permitiendo realizar en paralelo otras tareas. Además, el trabajar a la mitad de frecuencia de

trabajo permite optimizar el consumo de energía total del procesador, objetivo fundamental de una de las principales líneas de investigación del Grupo de Investigación GDEM.

6.2 Trabajos Futuros

A continuación se presentan una serie de trabajos futuros que tengan como punto de partida la integración realizada en este Trabajo Fin de Máster.

- Análisis de la caída de rendimiento del descodificador OpenSVC tras la integración realizada respecto a la versión de OpenSVC aislada y optimizada en [3].

- Paralelización de tareas entre el reproductor multimedia MPlayer y el descodificador OpenSVC ejecutados en 'ARM' y C64x+'respectivamente con el fin de optimizar el rendimiento de ambos núcleos.

- Análisis de la carga computacional del núcleo 'ARM' realizando la descodificación en el núcleo 'C64x+'.

- Estudio del descodificador OpenSVC para ver qué funciones y variables son más adecuadas para que estén alojados en memoria interna, de manera que el tiempo necesario para descodificar cada imagen se reduzca.

- Modificación de la aplicación para que sea posible cambiar la capa que se esté descodificando en tiempo real, ya que actualmente la capa a descodificar debe ser fijada antes de comenzar el proceso de descodificación. Esta línea de trabajo futuro propuesta ya se está llevando a cabo en un Proyecto Fin de Carrera dentro del GDEM.

- Medida del consumo de energía del procesador OMAP3530 para los diferentes escenarios planteados en esta memoria: descodificador OpenSVC ejecutado en el núcleo 'C64x+' a 360 MHz y en el 'ARM' a 720 MHz. Para la realización de esas medidas se hará uso de una tarjeta desarrollada en el proyecto PccMUTE (*Power Consumption Control in Multimedia TErminals*) con capacidad de medición en tiempo real.

7. REFERENCIAS BIBLIOGRÁFICAS

[1] Siavash Nesaii. "Adaptación de un reproductor de contenidos multimedia escalables para terminales móviles". Proyecto Fin de Carrera de la EUITT-UPM. Septiembre 2011.

[2] Fernando Pescador del Oso. "Contribución a las metodologías de optimización del tiempo de ejecución de algoritmos de descodificación de vídeo sobre DSPs". Tesis doctoral leída en el Departamento de Sistemas Electrónicos y de Control de la E.U.I.T de Telecomunicación de la U.P.M. Julio 2011.

[3] Ernesto Seisdedos Benzal. "Integración y optimización de un descodificador de vídeo escalable (H.264/SVC) para procesador OMAP3530". Trabajo Fin de Máster de la EUITT-UPM. Julio 2011.

[4] Texas Instruments. Procesador OMAP3530. "OMAP3530/25 Applications Procesor SPR8507F", Febrero 2008.
Disponible en http://www.ti.com/lit/ds/symlink/omap3530.pdf

[5] BeagleBoard. "Sitio web de BeagleBoard" 2011. http://beagleboard.org/

[6] ARM Cortex-A8. Arquitectura ARMv7. Más información disponible en http://www.arm.com/products/processors/cortex-a/cortex-a8.php

[7] Texas Instruments. "TMS320C64x/C64+ DSP CPU and Instruction Set SPRU732J" Julio 2010. Disponible en http://www.ti.com/lit/ug/spru732j/spru732j.pdf

[8] Controlador de salida de vídeo V4L2. Más información disponible en http://www.linuxfordevices.com/c/a/Linux-For-Devices-Articles/Intro-to-V4L2/

[9] Controlador de salida de vídeo Framebuffer. Más información disponible en http://www.linux-fbdev.org/HOWTO/index.html

[10] DigiKey Corporation. Más información disponible en http://www.digikey.es/

[11] BeagleBoard. "System Reference Manual Rev C4 BB_SRM". Disponible en http://beagleboard.org/static/BBSRM_latest.pdf

[12] Texas Instruments. "Powering OMAP™3 With TPS65950: Design-In Guide.SWCU056C". User's Guide. Octubre 2008. Disponible en http://www.ti.com/lit/ug/swcu056c/swcu056c.pdf

[13] uClinux™. Más información disponible en http://www.uclinux.org/

[14] Sistema operativo embebido Angström. Disponible en http://www.angstrom-distribution.org/

[15] Protocolo *Network File System*. Más información disponible en http://tools.ietf.org/html/rfc3530

[16] Texas Instruments. DSP/BIOS Link v1.65.00.03 para Linux. Mas información disponible en http://processors.wiki.ti.com/index.php/DSPLink_Overview

[17] Texas Instruments. CMEM. Mas información disponible en http://processors.wiki.ti.com/index.php/CMEM_Overview

[18] Texas Instruments. "TMS320 DSP/BIOS v5.41 SPRU423H" Agosto 2009. Disponible en http://www.ti.com/lit/ug/spru423h/spru423h.pdf

[19] Texas Instruments. Code Composer Studio v.5.1 para Linux. Disponible en processors.wiki.ti.com/index.php/Download_CCS

[20] Texas Instruments." DSP/BIOS 5.40 Textual Configuration (Tconf) User's Guide SPRU007L". Febrero 2009. Disponible en http://www.ti.com/lit/ug/spru007i/spru007i.pdf

[21] The Blackhawk USB560 JTAG emulator. Más información disponible en http://www.blackhawk-dsp.com/products/USB560.aspx

[22] Reproductor multimedia MoviePlayer. Más información disponible en http://www.mplayerhq.hu/design7/info.html

[23] Código fuente del reproductor multimedia MPlayer en lenguaje C disponible en http://www.mplayerhq.hu/design7/dload.html#source

[24] GTKPlayer. Widget de Mplayer para la reproducción de películas. Más información disponible en http://gtkplayerembed.sourceforge.net/

[25] ITU-T. Rec. H.264 "Advanced video coding for generic audiovisual services". Noviembre 2007. Scalable Video Coding (SVC).

[26] Fernando Pescador, David Samper, Matías J. Garrido, Eduardo Juárez, Mederic Blestel. "A DSP based SVC IP STB using OpenSVC Decoder". Consumer Electronics (ISCE), 2010 IEEE 14th International Symposium on, Germany (2010).

[27] Código fuente en lenguaje C que implementa un descodificador de vídeo compatible con el anexo G de la norma H.264. Disponible en http://sourceforge.net/projects/opensvcdecoder/

[28] Joint Scalable Video Model JSVM. Disponible en el repositorio de Rheinisch-Westfälische Technische Hochschule (RWTH) Aachen.

[29] Fernando Pescasdor, Mederic Blestel, Eduardo Juárez, Mickael Raulet, Matías J. Garrido. "H.264/SVC decoder performance comparison for DSP-Based consumer electronic applications". 14th IEEE International Symposium on Consumer Electronics ISCE 2011. Singapore, 14-17 Junio 2011.

[30] Codesourcery – Mentor Graphics Sourcery Tools. Más información disponible en http://www.mentor.com/embedded-software/codesourcery/

[31] Configuración *toolchain* ARM. GDEM intra. Disponible en http://gdem-intra.no-ip.org

[32] Texas Instruments. "DSP/BIOS™ LINK LNK 058 USR Version 1.65". User Guide. Disponible en http://software-dl.ti.com/dsps/dsps_public_sw/sdo_sb/targetcontent/DSPLink/1_65_01_06/exports/dsplink_linux_1_65_01_06.tar.gz

[33] David Casado Calvo. "Desarrollo de métodos de trabajo en la tarjeta BeagleBoard". Proyecto Fin de Carrera de la EUITT-UPM. Noviembre 2009.

[34] Texas Instruments. Code Composer Studio. "Importing projects". Disponible en http://processors.wiki.ti.com/index.php/Importing_Projects

[35] Texas Instruments. Code Composer Studio. "Debugging Projects v5". Disponible en http://processors.wiki.ti.com/index.php/GSG:Debugging_projects_v5

[36] Texas Instruments. Code Composer Studio. "How to Run GDB on CCSv5". Disponible en http://processors.wiki.ti.com/index.php/How_to_Run_GDB_on_CCSv5

[37] Texas Instruments. Code Composer Studio."Adding GEL file to targetconfiguration". Disponible en http://processors.wiki.ti.com/index.php/GSG:Adding_GEL_files_to_a_target_configuration_v5

[38] Texas Instruments. "DSP/BIOS™ LINK Programmer's Guide LNK 161 USR Version1.60".
Disponible en https://pixhawk.ethz.ch/_media/omap/programmersguide.pdf

[39] Espacio de color RG565. Más información disponible en http://www.imagingcontrol.com/en_US/support/documentation/class/PixelformatRGB565.html

[40] Texas Instruments. "Debugging the DSP side of a DSPLink application on OMAP using CCS". Dispobile en http://processors.wiki.ti.com/index.php/Debugging_the_DSP_side_of_a_DSPLink_application_on_OMAP_using_CCS

[41] YUV Player. Visor de ficheros en formato YUV. Disponible en www.elecard.com/en/products/professional/analysis/yuv-viewer.html

[42] Datahammer 7yuv. Visor de ficheros en formatos RGB y YUV. Disponible en http://datahammer.de/7yuvSetup-1.6.exe

[43] Software de edición Ultraedit. Versión de prueba disponible en http://www.ultraedit.com/downloads/ultraedit_download.html

[44] SCA- Based Open Source Software Defined Radio. "Installing DSPLink on a BeagleBoard Outside OpenEmbedded from Source". Disponible en http://ossie.wireless.vt.edu/trac/wiki/BeagleBoard_DSPLink

[45] Texas Instruments. "DSP/BIOS Timers and Benchmarking Tips SPRA829". Julio 2002. Disponible en http://www.ti.com/lit/an/spra829/spra829.pdf

[46] Texas Instruments. OMAP –L1 Linux Drivers Usage.Power Management. CPUFreq. Disponible en http://lxr.linux.no/linux+v2.6.32/+print=Documentation/cpu-freq/user-guide.txt

[47] MainConcept. SVC Baseline SDK DirectShow Documentation. Version 1.0.0. Aachen. Septiembre 2009.

[48] Fernando Pescador, Eduardo Juárez, Mikael Raulet, César Sanz. "A DSP Based H.264/SVC Decoder for a Multimedia Terminal". IEEE Transactions on Consumer Electronics on Consumer Electronics 2011. Aceptada para su publicación en el volumen de Mayo de 2011.

8. ACRÓNIMOS

Acrónimo	Inglés	Castellano
ALU	*Arithmetic Logic Unit*	Unidad Aritmético-Lógica
API	*Application Programming Interface*	Interfaz de programación de aplicaciones
ARM	*Advanced RISC Machine*	Arquitectura RISC Avanzada
CABAC	*Context-adaptive binary arithmetic coding*	Codificación Binaria Aritmética Adaptada al Contexto
CCS	*Code Composer Studio*	
CGT	*Code Generation Tools*	Herramientas de Generación de Código
CIF	*Common Intemediate Format*	
CMEM	*Contiguous* Memory Allocator	
CPU	*Central Processing Units*	Unidad Central de Procesamiento
DDS	*Display SubSystem*	Subsistema de Visualización
DMA	*Direct Access Memory*	Acceso a Memoria Directo
DDR	*Double Data Rate*	Doble Tasa de Transferencia
DSP	*Digital Signal Processsor*	Procesador Digital de Señal
DVI	*Digital Visual Interface*	Interfaz Visual Digital
EDMA3	*Enhanced Direct Memory Access v3.0*	Controlador de DMA mejorado
ES	*Elementary Stream*	Trama Elemental
FBDEV	*Framebuffer*	
FIFO	*First In, First Out*	Primero en entrar, primero en salir
FPS	*Frames per second*	Imágenes por segundo
GDB	*GNU Debugger*	
GEL	*General Extension Language*	Lenguaje de Extensión General
GOP	*Group of Pictures*	Grupo de Imágenes
GPMC	*General Propose Memory Controller*	Controlador de Memoria de Propósito General
GPP	*General Purpose Processor*	Procesador de Propósito General
GTK	*GIMP Tool Kit*	
HDMI	*High Definition Multimedia Interface*	Interfaz multimedia de alta definición
IDC	*Insulation-Displacement Connector*	Conector de Aislamiento por Desplazamiento
IDE	*Integrated Development Environment*	Entorno de desarrollo integrado
I2C	*Inter Integrated Circuit*	Bus de Comunicación entre Circuitos

IDMA	*Internal Direct Memory Access*	Controlador interno de DMA
IP	*Internet Protocol*	Protocolo de Internet
JSVM	*Joint Scalable Video Model*	
JTAG	*Joint Test Action Group*	
LCD	*Liquid Crystal Display*	Pantalla de Cristal Líquido
MAC	*Multiply and Accumulate* Unit	Unidad de sumas y multiplicaciones
MAR	*Memory Attribute Register*	Registro de máscaras de memoria
McBSP2	*Multi-Channel Buffered Serial Port 2*	Puerto Serie Multicanal con Capacidad de almacenamiento
MMC	*MultiMediaCard*	Tarjeta Multimedia
MMU	*Memory Management Unit*	Unidad de Gestión de Memoria
MPEG	*Movie Picture* Experts *Group*	Grupo de Expertos en Imágenes en Movimiento
MPlayer	*Movie Player*	Procesador con Instrucciones Complejas
MSGQ	*Messaging Queue*	
NAL	*Network Abstraction Layer*	Capa de Abstracción de Red
NFS	*Network File System*	Protocolo de Archivos en Red
NTSC	*National Television System Committee*	
OTG	*On-The-Go*	
PccMUTE	*Power Consumption Control in Multimedia TErminals*	Optimización del consumo de energía en terminales multimedia
PCB	*Printed Circuit Board*	Placa de Circuito Impreso
POP	*Package on Package*	Tipo de encapsulado
PPS	*Processing Parameter Set*	Parámetros Asociados a la Imagen
PRCM	*Power Reset and Clock Management*	Administración de Energía, Reset y Reloj
QDMA	*Quick Direct Memory Access*	Acceso rápido al controlador de DMA
RGB	*Red, Green, Blue*	Rojo, Verde, Azul
RISC	*Reduced Instruction Set Computer*	Procesador con Repertorio de Instrucciones Reducido
ROM	*Read-Only Memory*	Memoria de solo Lectura
RS232	*Recommended Standard 232*	Bus de comunicaciones RS232
RTOS	*Real Time Operating System*	Sistema Operativo en Tiempo Real
SD	*Secure Digital*	
SDL	*PC Simple Direct Media Layer Library*	

SDRAM	*Synchronous Dynamic Random Access Memory*	Memoria RAM Dinámica y Aleatoria
SDRC	*SDRam Controller*	Controlador de memoria RAM
SEI	*Supplemental Enhancement Information*	Información de Mejora Adicional
SNR	*Signal Noise Ratio*	Relación Señal a Ruido
SoC	*System On Chip*	Sistema en un Chip
SPI	*Serial Peripheral Interface*	
SPS	*Sequence Parameter Set*	Parámetros Asociados a la Secuencia
SVC	*Scalable Video Coding*	Codificación de Vídeo Escalable
TCP	*Transmission Control Protocol*	Protocolo de Control de Transmisión
UART3	*Universal Asynchronous Receiver-Transmitter*	Transmisor-Receptor Asíncrono Universal
UNIX	*UNiplexed Information and Computing System*	
USB	*Universal Serial Bus*	Bus Serie Universal
VCL	*Video Coding Layer*	Capa de Codificación de Vídeo
VLIW	*Very Long Instruction Word*	Procesador con Instrucciones Complejas